高等学校计算机专业规划教材

Python语言程序设计



王小银 王曙燕 孙家泽 编著

清华大学出版社

高等学校计算机专业规划教材

Python 语言程序设计

王小银　王曙燕　孙家泽　编著

清华大学出版社
北　京

内 容 简 介

本书以程序设计为主线，由浅入深、循序渐进地讲述 Python 语言的基本概念、基本语法和数据结构等基础知识。内容包括 Python 语言及其编程环境，数据类型、运算符和表达式，基本流程控制(顺序、选择和循环)，序列、字典与集合，函数与模块，文件，异常处理，面向对象程序设计，图形用户界面设计以及 Python 在数据挖掘中的应用。

本书注重实用性和实践性，通过对一些典型算法的解析及其实现给读者一些解题示范和启发；实例通俗易懂。

本书可作为高等学校 Python 语言程序设计课程的教材，也可作为工程技术人员和计算机爱好者的参考资料。

图书在版编目(CIP)数据

Python 语言程序设计/王小银，王曙燕，孙家泽编著. —北京：清华大学出版社，2017(2018.7 重印)
(高等学校计算机专业规划教材)
ISBN 978-7-302-48558-2

Ⅰ. ①P…　Ⅱ. ①王…　②王…　③孙…　Ⅲ. ①软件工具－程序设计－高等学校－教材
Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2017)第 241105 号

责任编辑：龙启铭　张爱华
封面设计：何凤霞
责任校对：焦丽丽
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：北京密云胶印厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：15.75　　**字　　数**：363 千字
版　　次：2017 年 12 月第 1 版　　**印　　次**：2018 年 7 月第2 次印刷
定　　价：39.00 元

产品编号：074684-01



前言

Python语言由荷兰人Guido van Rossum于1989年发明,1991年首次公开发行。Python语言经过二十多年的发展,已经广泛应用于计算机科学与技术、科学计算、数据的统计分析、移动终端开发、图形图像处理、人工智能、游戏设计、网站开发等领域。Python是一种面向对象、解释运行、扩展性很强的程序设计语言,语法简洁清晰,同时拥有功能丰富的标准库和扩展库。其标准库提供了系统管理、网络通信、文本处理、数据库接口、图形系统、XML处理等功能;扩展库则覆盖科学计算、Web开发、数据库接口、图形系统等多个领域,并且大多功能成熟而稳定。

通过Python语言程序设计课程的学习,读者可以掌握Python语言的程序结构、语法规则和编程方法,具有独立编写常规Python语言应用程序的能力,同时为设计大型应用程序和系统程序打下坚实的基础。本课程是数据结构、面向对象程序设计、操作系统和软件工程等课程的基础,可为这些课程提供实践工具。

本书以程序设计为主线,由浅入深、循序渐进地讲述Python语言的基本概念、基本语法和数据结构等基础知识。全书共13章,第1和第2章介绍了Python语言基本概念、基本数据类型、运算符和表达式;第3~5章介绍了三种基本程序设计结构(顺序结构、选择结构和循环结构);第6和第7章介绍了序列(包括列表、元组和字符串)、字典与集合;第8章介绍了函数与模块的定义和使用;第9和第10章介绍了文件、异常处理的基本知识;第11章介绍了面向对象程序设计相关知识及应用;第12章介绍了使用Python进行图形用户界面的设计;第13章介绍了Python在数据挖掘中的应用。本书中的例题均在Python 3.5运行环境中调试通过。

本书第1章由王曙燕编写,第2~12章及附录由王小银编写,第13章由孙家泽编写,全书由王小银统稿。研究生权雅菲和陈朋媛参与了部分校对工作,作者在此一并表示衷心的感谢。

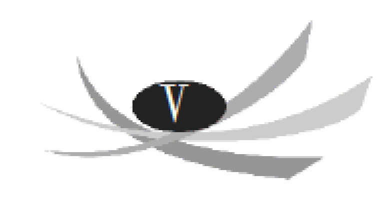
本书可作为高等学校Python语言程序设计课程的教材,也可作为工程技术人员和计算机爱好者的参考资料。

由于编者水平有限,书中难免存在不足之处,恳请广大读者批评指正。作者联系方式:wangxiaoyinxy@126.com。

编　者

2017年10月


目 录


V

第 1 章 Python 语言概述

Python 语言是一门面向对象的解释型高级程序设计语言，是一种新的脚本解释语言。Python 语言现在已发展成为一门功能强大的通用型语言。它不仅开源，而且支持命令式编程、面向对象程序设计和函数式编程，包含丰富且易理解的标准库和扩展库，可以快速生成程序的原型，帮助开发者高效地完成任务。Python 语言在设计上坚持清晰划一的风格，借鉴了简单脚本和解释语言的易用性，它的简洁、易读、易维护以及可扩展性，使其在科学计算领域受到广泛关注。Python 语言能够与多种程序设计语言完美融合，被称为胶水语言，能够实现多种编程语言的无缝拼接，充分发挥各种语言的编程优势。

1.1 Python 语言的发展

Python 自 1989 年推出至今已有二十多年的历史，其发展成熟且稳定。第一个 Python 编译器于 1991 年诞生，它既继承了传统语言的强大性和通用性，也具有脚本解释程序的易用性。只有你想不到，没有 Python 做不到。Python 宣告了一个新时代的开始。

1.1.1 Python 的起源

Python 继承于 ABC 语言，主要受到 Modula-3(Modula-3 是另一种相当优美且强大的语言，为小型团体所设计，并且结合了 UNIX shell 和 C 的习惯)的影响。发展初期，其发明者荷兰人 Guido 维护了一个 maillist(邮件列表开发方式)，Python 用户就通过邮件进行交流。Python 被称为 Battery Included，是说它标准库的功能强大。Python 的开发者来自不同领域，将不同领域的优点带给 Python。Python 具有开放性，易拓展，可以快速生成程序的原型。当用户不满足于现有功能时，可以根据具体项目对 Python 进行拓展或改造，并将新的特征加入到 Python 标准库中。Python 语言以对象为核心组织代码，支持多种编程范式，采用动态类型，自动进行内存回收。它支持解释运行，并能调用 C 库进行拓展，有强大的标准库。如 Python 标准库中的正则表达式参考 Perl，而 lambda、map、filter、reduce 等函数参考 Lisp。由于标准库的体系已经稳定，所以 Python 的生态系统开始拓展到第三方库。例如 Django、web. py、wxPython、Numpy、Matplotlib、PIL，将 Python 升级成了“物种”丰富的“热带雨林”。从 Python 2.0 开始，Python 也从 maillist 转变为完全开源的开发方式，获得了更加高速的发展。目前，Python 的框架已经确立，从 Python 2.x 到 Python 3.x，很多基本的函数接口都变了，有些库或函数甚至被删除或改名，第三方库支持 Python 2.x 的较多，支持 Python 3.x 的较少且不成熟。不过从长远来看，目前很多开源库开始支持 Python 3.x，所以 Python 3.x 任重道远，需要不断完善，慢慢被开发者接受。

1.1.2 Python的发展

随着互联网的发展,Python近几年在国内也被追捧。早在2011年1月的TIOBE编程语言排行榜上,Python就赢得2010年度语言的桂冠。与2010年同期比较,Python使用者增长了1.81%。因为Python的方便,越来越多的大学将Python作为一门教学课程。不仅如此,拥有高容量、高速度和多样性的大数据已成为当今时代的主流,移动互联网、云计算、大数据的快速发展,使Python为开发者带来巨大机会。Python作为一门设计优秀的程序语言,其开放、简洁和黏合,符合现发展阶段对大数据分析、可视化、各种平台程序协作具有快速促进作用的要求,大数据的火热和运维自动化必会带动Python的发展。Python能够帮助程序员完成各种开发任务,作为编制其他组件、实现独立程序的工具,必会在各种领域被广泛使用。Python自发布以来,虽一直饱受争议,但语言就像生命的进化,尤其是自由开源软件,替换是逐步的。总之,越来越多的人开始使用Python,使其晋升为当前的热门语言。

1.2 Python语言的特点

Python以语法清晰、结构简单、可读性高著称。Python的设计哲学是"优雅""明确""简单"。Python开发者的哲学是"用一种方法,最好是只有一种方法来做一件事"。Python代码通常被认为具备更好的可读性,并且能够支撑大规模的软件开发。Python关键字较少,而且没有像其他传统语言那样用来访问变量、定义代码块和进行模式匹配的命令式符号,如分号,使得Python代码语法清晰,易于阅读。Python良好的可读性使编程人员能够专注于解决问题而不是去搞明白语言本身,让人很容易理解开发者所写的代码。Python具有极其简单的说明文档,容易上手,很适合初学者。

1.2.1 Python的特性

1. 面向对象

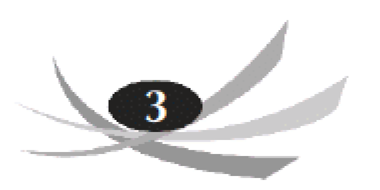
Python是完全面向对象的语言。面向对象编程支持将特定的功能与所要处理的数据相结合,即程序围绕着对象构建。例如,函数、模块、数字、字符串都是对象,并且完全支持继承、重载、派生、多继承,有益于增强代码的复用性。Python借鉴了多种语言的特性,支持重载运算符和动态类型。

2. 可移植性

Python已经被移植在许多平台上,是因为Python的解释器是用C语言编写的,而C语言的可移植性很好,使用Python开发的通用软件经过改动或不需任何改动就能够运行在任何平台上。这种可移植性既适用于不同的架构,也适用于不同的操作系统。

3. 解释性

Python是一种解释性语言,在开发过程中没有编译环节。用Python语言编写的程序不需要编译成二进制代码,可直接从代码运行程序。在计算机内部,Python解释器把



代码转换成近似机器语言的中间形式字节码，然后再把它翻译成计算机使用的机器语言并运行，使 Python 程序更简单、更加易于移植，从而改善了 Python 的性能。

4. 可扩展性

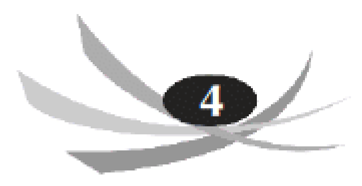
如果需要一段关键代码运行得更快或者希望某些算法不公开，可以部分程序用 C 或 C++ 语言编写，然后在 Python 程序中使用它们。Python 本身被设计为可扩充的，提供了丰富的 API 和工具，其标准实现是使用 C 语言完成的(CPython)，程序员能够轻松地使用 C、C++ 语言来编写 Python 扩充模块，缩短开发周期。Python 编译器本身也可以被集成到其他需要脚本语言的程序内，因此很多人还把 Python 作为一种"胶水语言"(glue language)使用，可以用 Python 将其他语言编写的程序进行集成和封装。Jython 是 Python 的 Java 实现，需用 Java 进行编写扩展；IronPython 是针对.NET 或 Mono 平台的 C# 实现，需用C# 或 VB.NET 来扩展 IronPython。PyRex 工具允许 C 和 Python 混合编程，这个工具会把所有的代码都转换成 C 语言代码，编写扩展很容易。

使用 Python 快速生成程序原型，有时甚至是程序的最终界面，然后对其中有特别要求的部分用更合适的语言改写，而后封装为 Python 可以调用的扩展类库。但是需要注意，在使用扩展类库时可能需要考虑平台问题，某些扩展类库可能不提供跨平台的实现。

5. 丰富的库

Python 有数百个标准库模块，包括 sys 模块、os 系统模块、re 模式匹配模块、字符串模块等。标准库可以帮助用户快速实现一些功能，不必重复开发已有的代码，可以提高效率和代码质量。标准库模块总是可用的，客户访问标准库模块时，必须使用 import 关键字将模块导入到客户模块中使用。模块包含与解释器相关的工具，也提供了对某些环境分量的访问，如命令行、标准流等。os 系统模块是 Python 3.x 和 Python 2.x 的主要操作系统(OS)的服务接口，它提供一般的 OS 支持和标准的独立于平台的 OS 工具集合。os 系统模块包括环境、文件、shell 命令等工具，也包括嵌入的子模块 os.path，这个子模块提供目录处理工具的便捷接口。re 模式匹配模块是 Python 3.x 和 Python 2.x 中标准正则表达式 pattern matching 的接口，正则表达式模式及由它们匹配的文本被指定为字符串。字符串模块为处理字符串对象定义常数和变量。其他标准库模块的详细信息请参阅 Python 标准库。Python 语言的核心只包含数字、字符串、列表、字典、文件等常见类型和函数，而 Python 标准库提供了文本处理、操作系统、网络通信、W3C 格式支持等额外的功能。Python 标准库命名接口清晰、文档良好，很容易学习和使用；通过基于标准库的大量工具，可以使用高级语言(如 C 和可以作为其他库接口的 C++)。只要安装了 Python，所有这些都能做到，这称为 Python 的"遥控器"哲学。

除了标准库，Python 还提供了大量高质量第三方库，可以在 Python 包索引中找到它们。Python 的第三方库使用方式与标准库类似，功能强大，提供了数据挖掘、大数据分析、图像处理等功能。近年来，由于 Python 库的不断发展，如 Pandas 库，Python 在数据挖掘领域崭露头角，在大数据分析和交互、探索性计算以及数据可视化等方面，相对于 R、MATLAB、SAS、Stata 等工具，Python 都有其优势。Pandas 库提供了能够快速、便捷地处理结构化数据的大量数据结构和函数，这些是使 Python 成为强大而高效的数据分析



环境的重要因素之一；Pandas 兼具 NumPy 高性能的数组计算功能以及电子表格和关系型数据库(如 SQL)灵活的数据处理功能，提供了复杂精细的索引功能，能更便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作。NumPy 是 Python 进行科学计算的基础包，在数据分析方面可作为在算法之间传递数据的容器，还可以将 C、C++ 、FORTRAN 代码集成到 Python 的工具。TVTK 是 Python 数据三维可视化库，它提供了 Python 风格的 API，并支持 Trait 库和 NumPy 数组。Trait 库可以为对象的属性添加检校功能，提高程序的可读性，降低出错率。Matplotlib 是最流行的用于绘制数据图表的 Python 库，提供了一种非常好用的交互式数据绘图环境。Scikit-Learn 是基于 Python 的机器学习库，建立在 NumPy、SciPy 和 Matplotlib 基础上，其操作简单，可进行高效的数据挖掘和数据分析。SciPy 是一组专门解决科学计算中各种标准问题域的包的集合。PIL 是基于 Python 的图像处理库，能够进行多格式图像处理。Python 的第三方库可以使用 Python 或者 C 语言编写，SWIG、SIP 常用于将 C 语言编写的程序库转化为 Python 模块。Boost C++ Libraries 包含了一组库 Boost. Python，使得以 Python 或 C++ 编写的程序能互相调用。Python 已成为一种强大的应用于其他语言与工具之间的“胶水”语言。

6. 易维护

代码维护是软件开发必不可少的组成部分。软件是一种商品，从软件开发出来的第一天开始，软件维护的问题也就应运而生。软件维护的工作量占据了软件生命周期的70%以上，因此，如何减少软件维护的工作量、降低软件维护成本、提高软件的可维护性是提高软件维护效率和质量的关键。Python 项目的成功很大程度上是因为 Python 代码的易于维护。

7. 健壮性

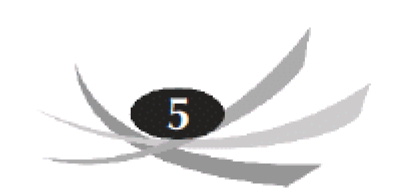
Python 提供了安全、合理的异常退出机制，能捕获程序的异常情况，允许程序员在错误发生的时候根据出错条件提供处理机制。一旦异常发生，Python 解释器会转出一个包含使程序发生异常的全部可用信息到堆栈并进行跟踪，此时程序员可以通过 Python 监控这些异常并采取相应措施。Python 的健壮性无论对用户还是软件设计人员都有诸多好处。

1.2.2 Python 的缺点

(1) 单行语句和命令行输出问题。很多时候不能将程序连写成一行，Python 必须将程序写入一个. py 文件。

(2) 独特的语法，强制缩进。这是 Python 语言的一大特色，也许不应该被称为局限，但是它用缩进来区分语句关系的方式还是给很多初学者甚至是有经验的程序员带来了困惑。最常见的情况是 tab 和空格的混用会导致错误，而这是用肉眼无法分别的。

(3) Python 在其 GIL 方面一直存在不足，但并非是致命缺点。GIL 是全局解释锁，Python 的多线程在多 CPU 条件下不能并行运行，只能在每个线程运行时先获得解释器的访问权限才能执行，而其他线程只能处于等待的状态，不过这一缺点可以通过多进程机制来弥补。



(4) Python 在某些领域性能相对较弱，对于性能要求非常高的项目，可以使用其他性能更好的语言实现和性能相关的那部分功能，再集成到 Python 内部。

1.2.3　Python 与其他语言的比较

Python 从诸多语言发展而来，拥有很多语言的特性，因此经常被人们拿来与其他语言进行比较。众所周知，Python 是一门解释性语言，进而与其他语言的比较是有针对性的，大多数比较是在 Perl、Java、Tcl 等语言之间。

Perl 设计之初就是为方便编写复杂、高效的系统脚本，是 Larry 为了格式化处理文本而创建，内建正则，是另一门广泛使用的脚本编程语言。Perl 对字符、文本文件处理能力很强，远远超越了标准的 shell 脚本，以前需要 shell＋sed＋awk＋C 才能完成的任务，现在只需 Perl 脚本即可。Perl 支持面向对象程序设计，正如 Python 一样，集所有编程语言所具有的功能特性于一身，也具备系统调用能力。Perl 擅长文字处理，其字符串匹配能力使 Perl 极具优势，它提供了一个强大的正则表达式引擎，可以对字符串文本加以过滤、识别和抽取。Python 的正则表达式引擎很大程度上基于 Perl。不过 Perl 对符号语法的过度使用，使程序的易读性很差，繁杂的语法使同一任务存在多个办法，程序员在项目开发过程中容易产生分歧，使开发周期延长。而 Python 是面向对象的动态通用语言，擅长数值处理，经常被用于脚本编程和快速开发，作为编译语言和脚本语言之间的桥接语言，语法简单，易读、易学、易懂，功能强大。可扩展性及面向对象的特征，使 Python 自然而然成为大规模应用程序开发工具。

Python 与 Java 有相似的面向对象特性和语法，但 Java 语法较为烦琐，而 Python 却以语法简洁著称，通常为开发人员提供更加快速的开发环境。在二者的关系上，我们自然而然会联想到 Jython，它是一种完整的语言，是 Python 语言在 Java 中的完全实现。Jython 可以直接调用 Java 的各种函数库，可以直接处理 Java 对象，在只有 JVM 的环境中运行 Python 程序。

Tcl 也是一种很热门的简单脚本语言，易扩展。Tcl 程序由一系列 Tcl 命令组成，主要用于给一些交互程序发布命令，在运行时由 Tcl 解释器解释运行。与 Python 相比，Tcl 也是一种解释型脚本语言，不过只有几种有限的数据类型，但可以通过创建新的过程以增强其内建命令的能力。由于 Python 有类、模块和包机制，更适合开发大型应用程序。

1.3　简单的 Python 程序介绍

本节将通过简单的 Python 程序实例，借助以前的编程经验，让读者更好地了解 Python 这门语言的基本结构，以及如何使用 Python 做一些简单的编程，引导读者快速入门。

在 Windows 系统命令行版本的 Python shell 的 Python 环境下(也就是 Command line 环境下)先启动 Python 解释器，需要注意的是，这里使用的是 Python 的 Command Line 环境，并非 Windows 的 cmd 环境。如同在普通文本中输入 Python 代码一样，下面用 print 语句完成第一个编程实例，输出“Hello World!”。

【例 1.1】 使用 Python 的 print 语句，在屏幕上输出一行文字“Hello World!”。

程序如下：

```
>>>print'Hello World!'
```

程序运行结果：

```
Hello World!
```

这就是我们编写的第一个 Python 程序，该程序非常简单，仅用一条 print 语句输出字符串“Hello World!”，且程序必须一行一行地输入，待程序输入完毕按 Enter 键(也称回车键)即可看到运行结果。本例中涉及 Python 的主提示符(>>>)，一旦主提示符出现，就表示 Python 解释器在等待用户输入下一条语句。Python 和诸多解释型脚本语言一样，使用语句进行输出。语句使用关键字来组成命令，Python 通过语句向解释器传达命令，限制 Python 的行为。语句通常可以有输出或者没有输出。Python 除了通过语句来达到用户的需求，也可以用表达式来实现，表达式包括函数、算数表达式等。表达式与语句的区别在于表达式没有关键字，而且表达式可以接受用户输入，也可以不接受用户输入；部分表达式有输出，有的则没有。

【例 1.2】 求 3 个数的平均值。

程序如下：

```
import math
sum=0
x,y,z=input('please input the number of x,y,z')
sum=x+y+z
aver=sum/3.0
print 'aver=',aver
输入数据：3,6,9
```

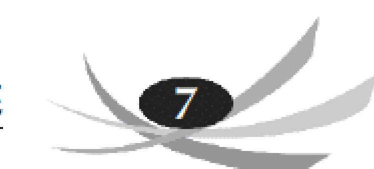
程序运行结果：

```
aver=6.0
```

本程序的作用是求 3 个数的平均数 aver，先求所有元素的总和，再将总和除以元素个数即可得到输入元素的平均值。该程序是在扩展名为.py 的文件中编写，再进行运行。程序的第 1 行是模块导入语句，导入一个模块后，就能使用该模块中的函数和类；第 2 行是为变量 sum 赋初值；第 3 行是接收从键盘输入的 3 个数；第 4 行是对键盘输入的 3 个数求和，这里也可以用 Python 的内建函数 sum()求和，关于 Python 的内建函数后续章节将会详细讲解；第 5 行是对 3 个数求平均值，这里我们看到 aver=sum/3.0，除以 3.0 只是为了让 aver 的最终结果输出为小数，否则默认输出为整数；第 6 行是输出语句，将 aver 的值输出到屏幕上。

【例 1.3】 创建一个由 5 个数构成的列表，由小到大进行升序排序。

程序如下：



```
score= [53,42,28,30,18]          #定义列表并给列表赋初始值
print('原始序列是:')
print(score)                     #打印列表初始值,也就是排序前的值序列
a= len(score)                    #求列表长度
for i in range(a- 1):            #循环变量 i 依次迭代列表[0,1,2,3]中的元素
min_score= score[i]              #将列表第 1 个数 score[i]赋给 min_score
m= i
for j in range(i+ 1,a,1):        #j 依次迭代列表[i+ 1,…,4]中的元素
    if min_score> sore[j]:       #找本轮所有待排序数中最小数
        min_score= score[j]      #本轮排序最小值存于 min_score
        m= j                     #本轮排序最小值下标存于 m
t= score[i]
score[i]= score[m]
score[m]= t                      #将本轮最小数与本轮序号第一的数进行交换
print('进行升序排序后的序列是:')
Print (score)
```

程序运行结果：

```
原始序列是:
[53,42,28,30,18]
进行升序排序后的序列是:
[18,28,30,42,53]
```

列表是 Python 语言中重要的内置数据类型，一个列表中的数据类型可以各不相同，列表元素间用逗号分隔，列表可以简化许多功能的实现。在这里需要注意的是在使用 for 循环和 if 条件语句时，Python 语法要求需要在其末尾加上冒号，而且 Python 通过空格控制语句缩进，注意 Tab 键和空格键不能混用。这里使用选择排序算法，其算法思想是：以升序为例，每轮排序过程中，选出本轮所有待排序数中最小数，然后将最小数与本轮序号第一的数进行交换；反复执行上述操作，直到所有元素排序完毕。排序算法使用双层循环嵌套结构，本题有 5 个数需要升序排序，一共排序 4 轮，因此外循环是 4 次，i 为外循环变量，每轮排序需经过若干次的比较才能找出本轮中的最小数，比较次数随着排序轮数的增加而减少；j 为内循环变量，内循环结束后，本轮排序选出的最小数即为 score[m]，将 score[m]与本轮序号第一的元素 score[i]进行交换。重复上述过程，经 4 轮排序后，就可以实现对列表中 5 个数的排序。

【例 1.4】　设计函数，求解圆形、长方形、三角形的面积。

程序如下：

```
#- * - coding:utf- 8- * -
#圆形面积
def CirArea(r):
    area= 3.14* r* r
    print("the area of circle is: ", area)
```

```
#长方形面积
def RectArea(a=5, b=6):
    area=a*b
    print("the area of rectangle is: ",area)
#三角形面积
def TriArea(b, h):
    area=1.0/2*b*h
    print("the area of triangle is: ")
    return area
CirArea(4)
RectArea()
print(TriArea(3,4))
```

程序运行结果：

```
the area of circle is:  50.24
the area of rectangle is:  20
the area of triangle is:
6.0
```

直接在 Python 代码中添加中文注释，在执行时会出现异常，未解决 Python 编码问题。可在源程序开始加上一行注释“#-*-coding:utf-8-*-”，即在 coding 与-*-之间，输入 Python 已知的编码方式，如 utf-8。程序中 def 表示一个函数的开始，后接函数标识符名称和括号，函数内容以冒号(:)开始，并且缩进。函数可以带参数和返回值，参数按从左向右匹配，每个参数也可设置默认值，若调用函数时没有传递参数，会按照默认值进行赋值。

1.4 Python 的程序开发工具

本节主要介绍 Python 开发工具，比较目前主流的两个 Python 版本，具体介绍 Python 的几种开发环境，读者根据自己需求安装所需版本。

1.4.1 Python 的版本选择

Python 自诞生到现在已经有多个版本，最主流的两个版本是 Python 2.x 和 Python 3.x。而这两个版本间有很大的差别，Python 3.x 在 Python 2.x 基础上做了许多改进与优化，包括许多基本的元素、函数的定义、接口和设计理念等。Python 3.x 在一定程度上并不能完全兼容 Python 2.x 版本。读者应根据需求来选择合适的版本。目前，企业使用最广泛的是 Python 2.x 版本，而对于学习研究而言，更多的选择 Python 3.x。Python 3.x 舍弃了一些 Python 2.x 中不再被维护的库，也必然是未来的发展趋势。

相较于 Python 2.x，Python 3.x 改变了许多函数接口，或对函数重新命名，甚至去掉了一些库和函数。相比较而言，Python 2.x 支持更多的第三方库，Python 3.x 支持的库相对不完善或者支持而不稳定。

本节主要从下面几个方面比较两个版本间的区别及变化。

1. print 语法

Python 2. x 和 Python 3. x 最突出的语法区别就是 print 语法。在 Python 2. x 中，print 是一条语句，输出内容跟在 print 关键字后面；在 Python 3. x 中，print()是个函数，输出内容作为 print 的参数。

下述程序为 Python 2. x 中 print 语句的输出，在 IDLE 中编写。在命令行中输入 IDLE 则启动集成编程窗口 IDLE，IDLE 窗口称为 Python shell，在提示符>>>后输入 print 语句，按 Enter 键后输出程序运行结果。

可输入一些简短的语句直接执行程序。

```
>>>print'hello Python2'
hello Python2
>>>print ('hello Python2')      #print 后面有空格
hello Python2
>>>print("hello Python2")       #print 后面不带有其他参数
hello Python2
>>>print("hello Python2","study")
('hello Python2','study')
```

下述程序为 Python 3. x 中 print()函数的输出。

```
>>>print('hello Python3')
hello Python3
>>>print('study','Python3')
study Python3
```

注意：三个大于号(>>>)是 Python 的主提示符，在提示符的后面输入程序语句，按 Enter 键运行程序，在 Python 界面中交互式动态显示程序结果。

2. Unicode 字符类型

Python 2. x 有 Unicode 和非 Unicode 两种字符串类型，对应的两个全局函数可分别将对象转换成字符串。其中，unicode()函数将对象强制转换成 Unicode 字符串，str()函数将对象转换成非 Unicode 类型；Python 3. x 只有 Unicode 一种字符串类型，相应地只有一种强制转换函数，即 str()函数。

3. 不等运算符

Python 2. x 不等运算符有两种：!＝和<>；

Python 3. x 只有一种写法：!＝。

4. 数据类型

Python 2. x 有两种整形数据类型：long 和 int 类型；Python 3. x 保留了 int 类型，舍

弃了 long 类型，并将原来的 8 位字符串(str 类型)改为二进制数据(bytes 类型)，新增了 bytes 数据类型。

下述程序为 Python 3. x 版本中定义 bytes 数据类型的语句。

```
>>> bt=b'Python'              #定义 bytes 数据类型
>>> type(bt)                  #输出变量类型函数
<class 'bytes'>               #输出类型为 bytes 类型
```

使用 str. encode()和 bytes. decode()函数，Python 3. x 可在文本字符串(str)与二进制(bytes)数据之间进行两种数据类型的相互转换。下述程序为 Python 3. x 版本中数据类型的转换语句。

```
>>> str= 'Python,Battery Include'
>>> type(str)
<class 'str'>
>>> b= str.encode('utf- 8')
>>> print(b)
b'\xe4\xba\xba\xe7\x94\x9f\xe8\x8b\xa6\xe7\x9f\xad\xef\xbc\x8c\xe6\x88\x91\xe7\x94\xa8Python'
>>> c=b.decode('utf- 8')
>>> print(c)
Python,Battery Include
```

Python 版本间的区别还体现在其他方面，本节不再赘述，读者可从 Python 官方网站提供的资料中了解更多、更详细的内容，官方网站地址为 https://docs. Python. org/3/whatsnew/3. 0. html#overview-of-syntax-changes。

1.4.2 Python 的安装

Linux 系统一般都会自带 Python 开发环境，读者可根据自己的需求在 Python 官方网站下载安装所需版本，安装过程与其他软件安装过程类似，本节不再详述。安装完成后，需检验是否安装成功。在命令行中输入语句：Python -V 可以查看 Python 版本，这里的 V 需要大写。若安装成功，可看到相应的 Python 版本信息，如下所示。

```
[root@ localhost cpy]          Python - V
Python 3.3.0
```

1.4.3 Python 的开发环境

Python 有多种开发环境，本节主要介绍 Python 的原始开发、交互式开发以及集成开发环境。

1. 原始开发环境

Python 最原始的开发方式是使用合适的文本编辑器创建一个. py 文件，文件中写入

Python 程序，保存文件并在命令行终端环境下运行这个 Python 脚本（.py 文件）。

1）选择合适的编辑器

Python 代码本身就是文本文件，因此只需要选择合适的文本编辑器，保存文本为.py 文件，即可在命令行终端环境下运行这个 Python 脚本。一般情况下，Windows 系统选择使用 Notepad＋＋编辑器，Linux 系统推荐使用 Sublime 编辑器。

Notepad＋＋适合于 Windows 系统中文本编辑器，其功能比 Windows 系统自带的 Notepad（记事本）编辑器的功能强大。Notepad＋＋内置的多种语法呈高亮显示，可自动检测文件类型，支持自定义语言。但是，Notepad＋＋只支持 Windows 平台。

Sublime 界面美观，功能强大，是一款跨平台的文本编辑器，支持多种操作系统，如 Windows、Mac、Linux 等。Sublime 可实现多种编程语言的语法高亮显示，可实现代码自动完成，拥有良好的扩展能力和用户自定义配置功能。

2）在命令行终端运行 Python 脚本

打开操作系统命令行终端，切换当前路径到脚本文件所在位置，在当前路径下输入脚本文件名（.py 文件），按 Enter 键运行该脚本文件。

2. 交互式开发环境

Python 中的 shell 即 interactive shell，是 Python 的交互式运行环境，在该环境下运行程序可动态、交互地显示程序动态运行结果。Python 提供了两种交互式开发环境：命令行版的 Python shell 和带图形界面的 Python IDLE。

1）命令行（command line）版本的 Python shell

命令行版本的 Python shell 在 Python 软件安装成功后即可直接使用，不需要另外重新安装。Linux 系统中，在终端输入 python 可直接打开命令行 Python shell，图 1.1 为打开的 Python shell 窗口。



图 1.1　Python shell 窗口

打开 Python shell 即可看到当前的 Python 版本信息及系统信息，在三个大于号（＞＞＞）提示符后面编写 Python 代码。该窗口可动态、交互地显示代码相应信息。

2）带图形界面的 Python IDLE

Python IDLE（Python GUI）是相对于 command line 版的 Python shell，是在其基础上加上图像化显示，界面更加友好，也可自动显示对象的属性和方法。在 Windows 系统下安装 Python 后，自身就带有 IDLE。Linux 系统在安装的 Python 软件中，一般不会带有 IDLE 窗口，开发人员若有需要，可自行在官方网站下载安装，这里不再叙述。

3）在 IDLE 窗口编写 Python 程序

（1）创建 Python 文件并保存：打开 IDLE，在该窗口中可直接输入代码并运行。另外，也可首先创建一个窗口。创建方式：在菜单栏选择 File→New File，或者按组合键 Ctrl＋N，在创建的 IDLE 窗口中编写程序，保存文件为.py 文件。图 1.2 为新创建的 Python 文件。

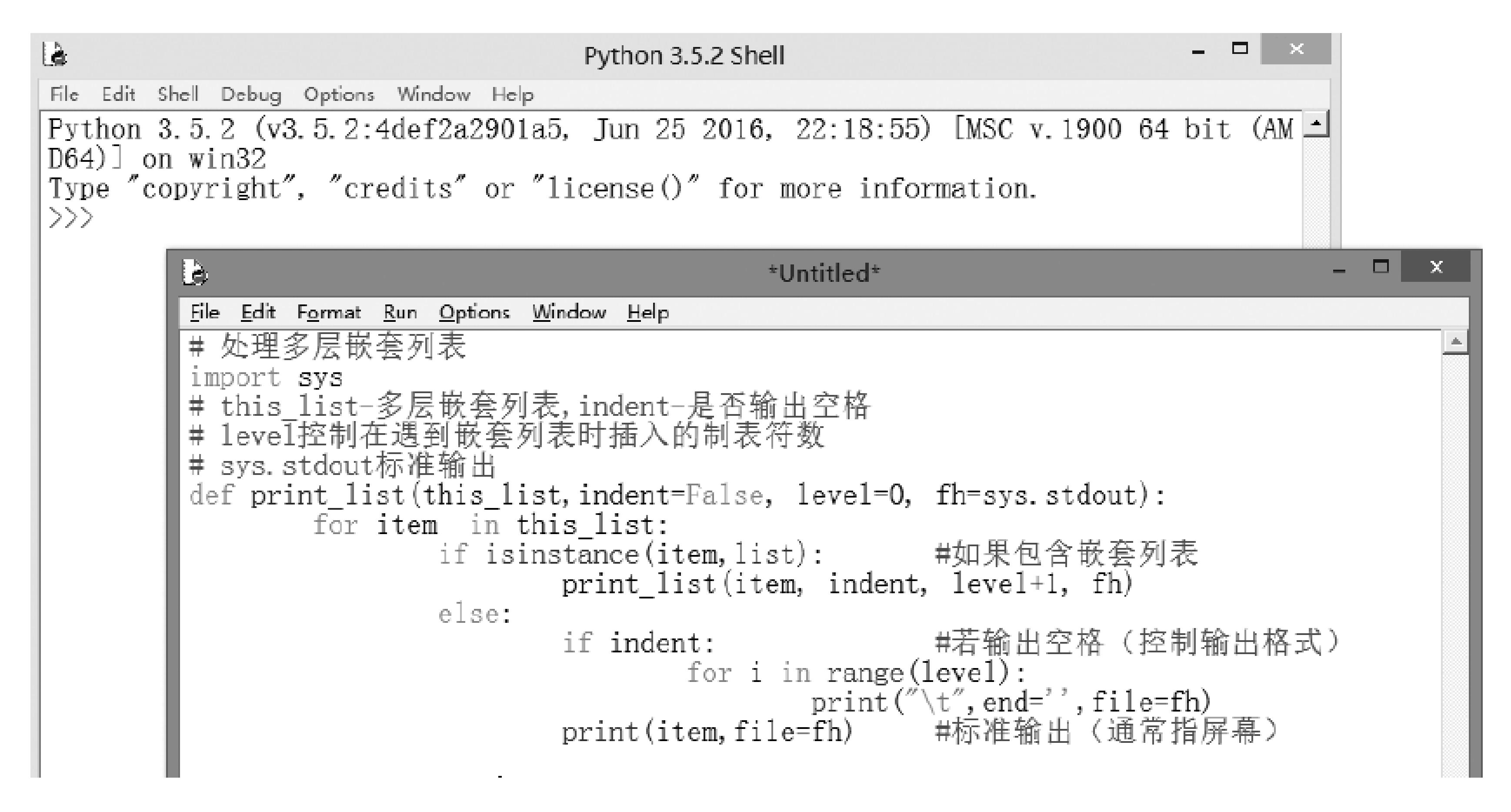


图 1.2　新创建的 Python 文件

注意：

① Python 代码需要严格按照格式书写，代码中的空格不仅是为了美观，更重要的是靠空格来控制代码格式。

② IDLE 支持代码复制，但如果一次性直接赋值多行代码，一般都会出错，建议按照 Python 格式，一行一行地复制。

（2）运行 Python 程序：保存 Python 文件后，在 IDLE 窗口菜单栏选择 Run→Run Module 或按快捷键 F5，即可运行程序。图 1.3 为图 1.2 示例程序的运行结果。

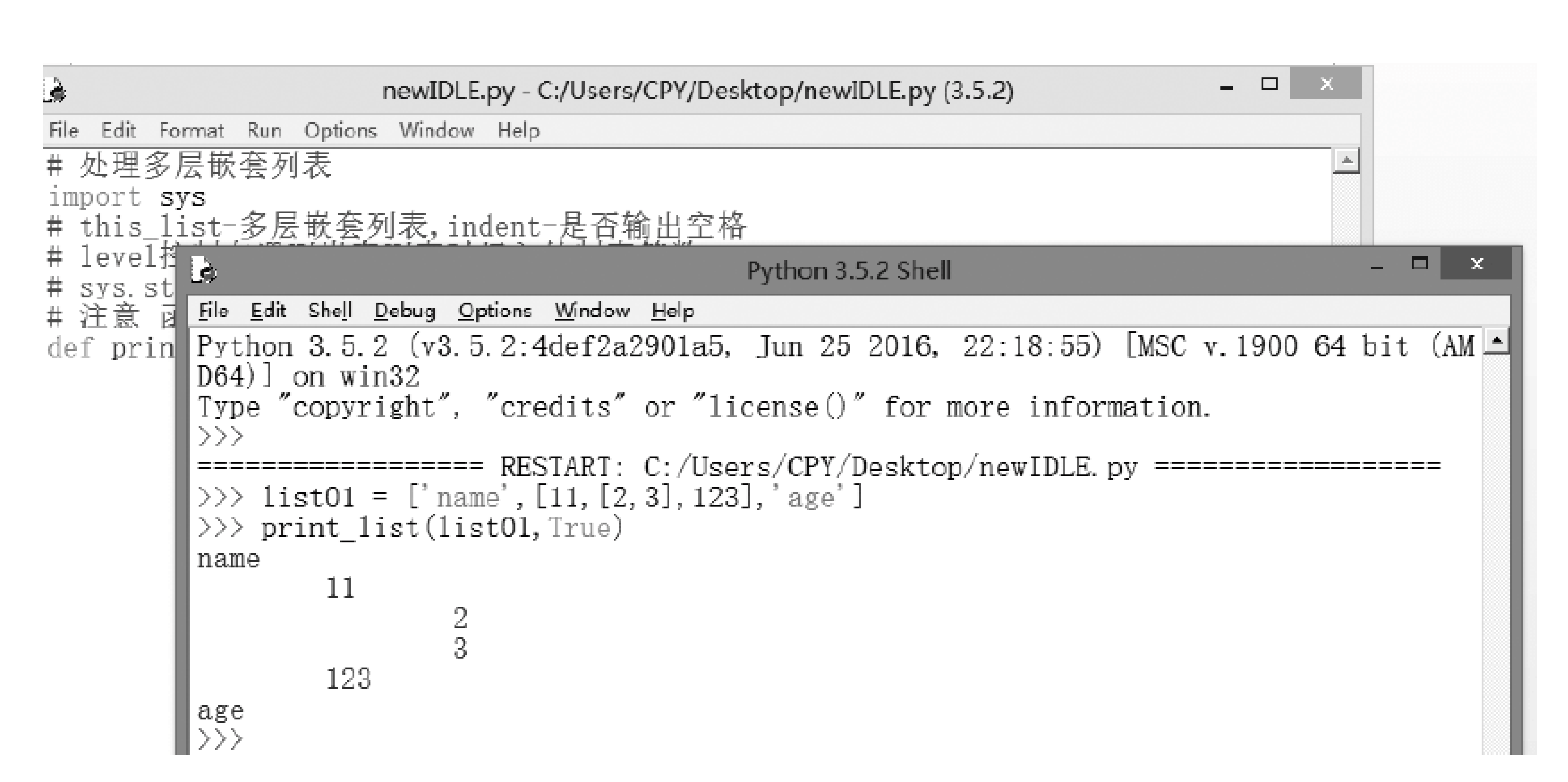


图 1.3　Python 程序运行结果

3. 集成开发环境

集成开发环境(Integrated Development Environment,IDE)是将相对独立的软件集成到一起,为程序开发提供更好的编程环境。Python shell 开发环境适合于编写简单的代码,若编写复杂的项目,在 Python shell 环境中效率并不高,代码调试也相对困难。Python IDE 即 Python 的集成开发环境,是将 Python 开发相关的多种工具集成在一起,同时可实现代码、文件、项目等的组织和管理等功能。

常见的 Python IDE 工具有 PyScripter、Eclipse＋PyDev、Ulipad、Pycharm、Eric 等。本节不再详细说明,感兴趣的读者可查阅相关资料进一步了解。

习　　题

1. 与其他程序设计语言相比 Python 有哪些特点?
2. 输入三个数,按从大到小的顺序输出。
3. 编写程序,输出以下信息:

```
***************
How are you!
***************
```

第2章 数据类型、运算符和表达式

程序处理的对象是数据，编写程序也就是对数据的处理过程，而运算符是对数据进行处理的具体描述。要学好 Python 并使用 Python 来编程，必须熟练掌握 Python 中的数据类型描述以及运算符与表达式，这是学习 Python 的重要基础，后续章节的内容都是在此基础上展开的。

2.1 常量、变量与标识符

Python 中存在两种表示数据的形式：常量和变量。常量用来表示数据的值；变量不仅可以用来表示数据的值，而且可以用来存放数据，因为变量对应着一定的内存单元。常量和变量都需要有一个名字（即标识符）来表示，因此本节首先介绍标识符及其命名规则。

2.1.1 标识符

标识符在程序中是用来标识各种程序成分，命名程序中的一些实体，如变量、常量、函数等对象的名字。

Python 规定，合法的标识符是由字母、数字和下画线组成的序列，且必须由字母或下画线开头，自定义的标识符不能与关键字同名。

以下是合法的标识符。

```
x,a1,wang,num_1,radius,_1, PI
```

以下是不合法的标识符。

```
a.1,1sum,x+y, !abc, 123,π, 3-c
```

在 Python 中，大写字母和小写字母被认为是两个不同的字符，因此标识符 SUM 与标识符 sum 是不同的标识符。习惯上符号常量名用大写字母表示，变量名用小写字母表示。

在 Python 中，单独的下画线（_）用于表示上一次运算的结果。例如：

```
>>> 20
20
>>>_* 10
200
```

标识符的命名规则如下。

(1) 变量名和函数名中的英文字母一般用小写,以增加程序的可读性。

(2) 见名知义:通过变量名就知道变量值的含义。一般选用相应英文单词或拼音缩写的形式,如求和用 sum,而尽量不要使用简单的代数符号,如 x、y、z 等。

(3) 尽量不要使用容易混淆的单个字符作为标识符,例如数字 0 和字母 o,数字 1 和字母 l。

(4) 应该避免开头和结尾都使用下画线的情况,因为 Python 中大量采用这种名字定义了各种特殊方法和变量。

关键字又称保留字,是 Python 语言中用来表示特殊含义的标识符,由系统提供,是构成 Python 语法的基础。

关键字不允许另作他用,否则执行时会出现语法错误。

可以在使用 import 语句导入 keyword 模块之后,使用 print(keyword. kwlist)语句查看所有 Python 关键字。在 Python 3. x 中共有 33 个关键字。查看关键字的语句如下。

```
>>> import keyword
>>> print(keyword.kwlist)
 ['False','None','True','and','as','assert','break','class','continue','def','del','elif','else','except','
finally','for','from','global','if','import','in','is','lambda','nonlocal','not','or','pass','raise','return
','try','while','with','yield']
```

另外还可以使用 keyword. iskeyword (word)的方式查看 word 是否为关键字。

```
>>> keyword.iskeyword ('False')
True                          #判断结果输出,'False'是 Python 的关键字
>>> keyword.iskeyword ('int')
False                         #判断结果输出,'int'不是 Python 的关键字
```

2.1.2　常量

在程序运行过程中,其值不能改变的量称为常量。在基本数据类型中,常量按其值的表示形式可分为整型、实型、字符型、布尔型和复数类型。例如: −123、20 是整型常量,3. 14、0. 15、−2. 0 是实型常量,'Python'、"Very Good!"字符串常量,True 是布尔型常量、3+2. 5j 是复数类型常量。

2.1.3　变量

在 Python 中,不需要事先声明变量名及其类型,类型是在运行过程中自动决定的,直接赋值即可创建各种类型的变量。变量在程序中使用变量名表示。变量名必须是合法的标识符,并且不能使用 Python 关键字。Python 是动态类型的语言,也是强类型语言(只能对一个对象进行适合该类型的有效操作)。Python 中每个对象包含三个基本要素,分别是 id(身份标识)、type(数据类型)和 value(值)。

例如:

```
>>> x=5
```

创建了整型变量 x,对其赋值为 5,如图 2.1 所示。



图 2.1　变量连接到对象

对该语句 Python 将会执行三个步骤去完成这个请求,这些步骤反映了 Python 语言中所有赋值的操作。

(1) 创建一个对象来代表值 5;

(2) 创建一个变量 x,如果它还没有创建的话。

(3) 将变量与新的对象 5 相连接。

在 Python 中从变量到对象的连接称为引用。Python 的赋值将在 2.3.3 节详细介绍。

```
>>> type(x)
<class'int'>
```

采用内置函数 type()返回变量 x 的类型 int。

```
>>> string="Hello World!"
```

创建了字符型变量 string,并赋值为 Hello World!。

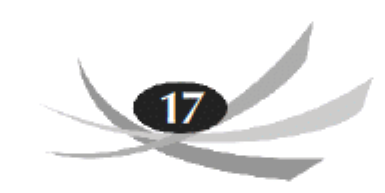
这种方法适合于任意类型的对象。

虽然 Python 不需要事先声明变量名及其类型,但 Python 仍属于强类型编程语言,其解释器会根据赋值或者运算来自动推断变量的类型。每种类型支持的运算也不完全相同,因此在使用变量时需要程序员自己确定所进行的运算是否合适,以免出现异常或者意料之外的结果。

注意:Python 是一种动态类型语言,即变量的类型可以随时变化。例如:

```
>>> x=5
>>> type(x)
<class'int'>
>>> x="Hello World!"
>>> type(x)
<class'str'>
>>> x=(1,2,3)
>>> type(x)
<class'tuple'>
```

首先通过赋值语句“x=5”创建了整型变量 x,然后又分别通过赋值语句“x="Hello World!"”和“x=(1,2,3)”创建了字符串和元组类型的变量 x。当创建了字符串类型变量 x 之后,之前创建的整型变量 x 就自动失效了,创建了元组类型变量 x 之后,之前创建的字符串类型变量 x 就自动失效了。也就是在显式修改变量类型或者删除变量之前,变



量将一直保持上次的类型。

2.2　Python 的基本数据类型

数据是计算机程序加工处理的对象。抽象地说,数据是对客观事物所进行的描述,而这种描述是采用了计算机能够识别、存储和处理的形式进行的。程序所能够处理的基本数据对象被划分成一些集合。属于同一集合的各数据对象都具有同样的性质,例如,对它们能做同样的操作,它们都采用同样的编码方式等。把程序中具有这样性质的集合称为数据类型。

在程序设计的过程中,计算机硬件也把被处理的数据分成一些类型。CPU 对不同的数据类型提供了不同的操作指令,程序设计语言中把数据划分成不同的类型也与此有着密切的关系。在程序设计语言中,都是采用数据类型来描述程序中的数据结构、数据的表示范围和数据在内存中的存储分配等。可以说数据类型是计算机领域中一个非常重要的概念。

Python 的数据类型如图 2.2 所示。Python 提供了一些内置的数据类型,它们由系统事先预定义,在程序中可以直接使用。本节主要介绍 Python 中简单类型的应用。

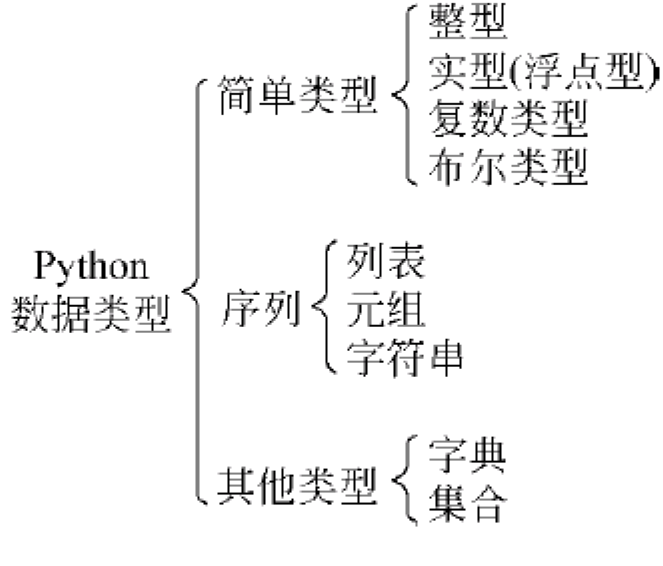


图 2.2　Python 的数据类型

221　整型数据

整型数据即整数,不带小数点,可以有正号或者负号。在 Python 3.x 中,整型数据在计算机内的表示没有长度限制,其值可以任意大。

例如,下面数据的表示在 Python 中是正确的。

```
>>> a= 12345678900123456789
>>> a * a
152415787504953525750053345778750190521
```

Python 中整型常量可用十进制、二进制、八进制和十六进制 4 种形式表示。

(1) 十进制整数。由 0～9 的数字组成,如－123,0,10,但不能以 0 开始。

以下各数是合法的十进制整常数:237,－568,1627。

以下各数是不合法的十进制整常数:023(不能有前缀 0),35D(不能有非十进制数码 D)。

(2) 二进制常数。以 0b 为前缀,其后由 0 和 1 组成。如 0b1001 表示二进制数 1001,即$(1001)_2$,其值为 $1\times2^3+0\times2^2+0\times2^1+1\times2^0$,即十进制数 9。

以下各数是合法的二进制数:0b11(十进制为 3),0b111001(十进制为 57)。

以下各数是不合法的二进制数:101(无前缀 0b),0b2011(不能有非二进制码 2)。

(3) 八进制整数。以 0o 为前缀,其后由 0～7 的数字组成,如 0o456 表示八进制数 456,即$(456)_8$,其值为 $4\times8^2+5\times8^1+6\times8^0$,即十进制数 302;－0o11 表示八进制数－11,即十进制数－9。

以下各数是合法的八进制数:0o15(十进制为 13),0o101(十进制为 65),0o0177777

(十进制为 65535)。

以下各数是不合法的八进制数：256(无前缀 0)，0o283(不能有非八进制码 8)。

(4) 十六进制整数。以 0x 或 0X 开头，其后由 0～9 的数字和 a～f 的字母或 A～F 的字母组成，如 0x7A 表示十六进制数 7A，即 $(7A)_{16}=7\times16^1+A\times16^0=122$；$-$0x12 即十进制数$-$18。

以下各数是合法的十六进制数：0x1f(十进制为 31)，0xFF(十进制为 255)，0x201(十进制为 513)。

以下各数是不合法的十六进制数：8C(无前缀 0x)，0x3H(含有非十六进制数码 H)。

注意：在 Python 中是根据前缀来区分各种进制数的，因此在书写常数时不要把前缀弄错，以免造成结果不正确。

【例 2.1】 整型常量示例。

```
>>> 0xff
255
>>> 2017
2017
>>> 0b10011001
153
>>> 0b012
SyntaxError: invalid syntax
>>> - 0o11
- 9
>>> 0xfe
254
```

222 实型数据

实数又称浮点数，一般有以下两种表示形式。

(1) 十进制小数形式。由数字和小数点组成(必须有小数点)，如 1.2、.24、0.0 等。浮点型数据允许小数点后没有任何数字，表示小数部分为 0，如 2. 表示 2.0。

(2) 指数形式。用科学计数法表示的浮点数，用字母 e(或 E)表示以 10 为底的指数，e 之前为小数部分，之后为指数部分，如 123.4e3 和 123.4E3 均表示 123.4×10^3。用指数形式表示实型常量时要注意 e(或 E)前面必须有数字，后面必须是整数，如 15e2.3、e3 和 .e3 都是错误的指数形式。

一个实数可以有多种指数表示形式，例如 123.456 可以表示为 123.456e0、12.3456e1、1.23456e2、0.123456e3 和 0.0123456e4 等多种形式。把其中 1.23456e2 称为规范化的指数形式，即在字母 e 或 E 之前的小数部分中，小数点左边的部分应有且只有一位非零的数字。一个实数在用指数形式输出时，是按规范化的指数形式输出的。

对于实型常量，Python 3.x 默认提供 17 位有效数字的精度，相当于 C 语言中双精度



浮点数。例如：

```
>>> 1234567890012345.0
1234567890012345.0
>>> 12345678900123456789.0
1.2345678900123458e+19
>>> 15e2
1500.0
>>> 15e2.3
SyntaxError: invalid syntax
```

223　字符型数据

在 Python 中定义一个字符串可以用一对单引号、双引号或者三引号进行界定，且单引号、双引号和三引号还可以相互嵌套，用于表示复杂的字符串。例如：

```
>>> "Let's go"
"Let's go"
>>> s="'Python' Program"
>>> s
"'Python' Program"
```

用单引号或双引号引起来的字符串必须在一行内表示，用三引号引起来的字符串可以是多行的。例如：

```
>>> s='''
'Python' Program
'''
>>> s
"\n'Python' Program\n"
```

除了以上形式的字符数据外，对于常用的但却难以用一般形式表示的不可显示字符，Python 语言提供了一种特殊形式的字符常量，即用一个转义标识符“\”(反斜线)开头的字符序列，如表 2.1 所示。

表 2.1　Python 常用的转义字符及其含义

字符形式	含　　义
\n	回车换行，将当前位置移到下一行开头
\t	横向跳到下一制表位置(Tab)
\b	退格，将当前位置退回到前一列
\r	回车，将当前位置移到当前行开头

续表

字符形式	含　　义
\f	走纸换页，将当前位置移到下页开头
\\	反斜线符“\”
\'	单引号符
\"	双引号符
\ddd	1～3 位八进制数所代表的字符
\xhh	1～2 位十六进制数所代表的字符

使用转义字符时要注意以下几点：

(1) 转义字符多用于 print()函数中。

(2) 转义字符常量，如'\n'，'\x86'等只能代表一个字符。

(3) 反斜线后的八进制数可以不用 0 开头。如'\101'代表字符常量'A'，'\134'代表字符常量'\'。

(4) 反斜线后的十六进制数只能以小写字母 x 开头，不允许用大写字母 X 或 0x 开头。

【例 2.2】 转义字符的应用。

```
a=1
b=2
c='\101'
print("\t%d\n%d%s\n%d%d\t%s"%(a,b,c,a,b,c))
```

程序运行结果：



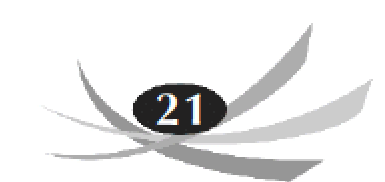
```
□□□□□□□□□1
2A
12□□□□□□A
```

在 print()函数中，首先遇到第一个“\t”，它的作用是让光标到下一个制表位置，即光标往后移动 8 个单元，到第 9 列，然后在第 9 列输出变量 a 的值 1。接着遇到“\n”，表示回车换行，光标到下行首列的位置，连续输出变量 b 和 c 的值 2 和 A，其中使用了转义字符常量'\101'给变量 c 赋值。遇到“\n”，光标到第 3 行的首列，输出变量 a 的值 1 和 b 的值 2，再遇到“\t”光标到下一个制表位即第 9 列，然后输出变量 c 的值 A。

224　布尔型数据

布尔类型数据用于描述逻辑判断的结果，有真和假两种值。Python 的布尔类型有两个值：True 和 False(注意要区分大小写)，分别表示逻辑真和逻辑假。

值为真或假的表达式为布尔表达式。Python 的布尔表达式包括关系表达式和逻辑表达式，它们通常用来在程序中表示条件，条件满足时结果为 True，条件不满足时结果为 False。



【例 2.3】 布尔型数据示例。

```
>>> type(True)
<class 'bool'>
>>> True==1
True
>>> True==2
False
>>> False==0
True
>>> 1>2
False
>>> False>-1
True
```

布尔型数据还可以与其他数据类型进行逻辑运算。Python 规定：0、空字符串、None 为 False，其他数值和非空字符串为 True。例如：

```
>>> 0 and False
0
>>> None or True
True
>>> "" or 1
1
```

225 复数类型数据

Python 支持相对复杂的复数类型。复数由两部分组成：实部和虚部。复数的形式为：实部＋虚部 j，例如 2＋3j。数末尾的 j(大写或者小写)表明它是一个复数。例如：

```
>>> x=3+5j                   #x为复数
>>> x.real                   #查看复数实部
3.0
>>> x.imag                   #查看复数虚部
5.0
>>> y=6-10j                  #y为复数
>>> x+y                      #复数相加
(9-5j)
>>> x-y                      #复数相减
(-3+15j)
>>> x*y                      #复数相乘
(68+0j)
>>> x/y                      #复数相除
```

```
(-0.23529411764705885+0.4411764705882353j)
```

2.3 运算符与表达式

2.3.1 Python运算符

Python 语言提供了丰富的运算符和表达式，这些丰富的运算符使 Python 语言具有很强的表达能力。

1. 运算符

Python 语言的运算符按照它们的功能可分为以下几种。

(1) 算术运算符(+、-、*、/、**、//、%)。

(2) 关系运算符(>、<、>=、<=、==、!=)。

(3) 逻辑运算符(and、or、not)。

(4) 位运算符(<<、>>、~、|、^、&)。

(5) 赋值运算符(=、复合赋值运算符)。

(6) 成员运算符(in、not in)。

(7) 同一运算符(is、is not)。

(8) 下标运算符([ ])。

(9) 其他(如函数调用运算符())。

若按其在表达式中与运算对象的关系(连接运算对象的个数)可分为如下几种。

(1) 单目运算符(一个运算符连接一个运算对象)：!、~、++、− −、−(取负号)、*、&、sizeof、(类型)。

(2) 双目运算符(一个运算符连接两个运算对象)：+、−、*、/、%、>、<、==、>=、<=、!=、&&、||、<<、>>、|、^、&、=、复合赋值运算符。

(3) 其他：()、[ ]、·、->。

2. Python 运算符的优先级和结合性

Python 中的运算符具有一般数学运算的概念，即具有优先级和结合性(也称结合方向)。

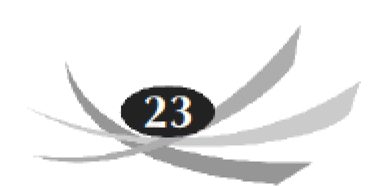
(1) 优先级：指同一个表达式中不同运算符进行运算时的先后次序。通常所有单目运算的优先级高于双目运算。

(2) 结合性：指在表达式中各种运算符优先级相同时，由运算符的结合性决定表达式的运算顺序。它分为两类：一类运算符的结合性为从左到右，称为左结合性；另一类运算符的结合性是从右到左，称为右结合性。通常单目、三目和赋值运算符是右结合性，其余均为左结合性。

关于 Python 中运算符的优先级见附录 B。

3. 表达式

表达式就是用运算符将操作数连接起来所构成的式子。操作数可以是常量、变量和



函数。各种运算符能够连接的操作数的个数、数据类型都有各自的规定，要书写正确的表达式就必须遵循这些规定。例如，下面是一个合法的 Python 表达式。

```
10+ 'a'+ d/e- i* f
```

每个表达式不管多么复杂，都有一个值。这个值是按照表达式中运算符的运算规则计算出来的结果。求表达式的值是由计算机系统来完成的，但程序设计者必须明白其运算步骤、优先级、结合性和数据类型转换这几方面的问题，否则就得不到正确的结果。

2.3.2　算术运算符和算术表达式

1. 算术运算符

算术运算符用于各类数值运算，包括+、—、*、/、//、* * 和%七种。基本算术运算符属性如表 2.2 所示。

表 2.2　基本算术运算符属性

运算符	含　义	优　先　级	结　合　性
+	加法	这些运算符的优先级相同，但比下面的运算符优先级低	左结合
—	减法		
*	乘法	这些运算符的优先级相同，但比上面的运算符优先级高	
/	除法		
//	整除		
* *	幂运算		
%	取模		

Python 中除法有两种：/和//，在 Python 3. x 中分别表示除法和整除运算。例如：

```
>>> 3/5
0.6
>>> 3//5
0
>>> 3.0/5
0.6
>>> 3.0//5
0.0
>>> - 3.0/5
- 0.6
>>> 3.0/- 5
- 0.6
>>> 3.0//- 5
- 1.0
```

```
>>>-3.0//5
-1.0
```

Python中很多运算符有多重含义，在程序中运算符的具体含义取决于操作数的类型。例如：*运算符在数值型数据中进行运算表示乘法；对于序列类型，如列表、元组、字符串运算，则表示对内容进行重复。具体的用法将在后续章节进行介绍。

```
>>>3*5                    #整数相乘运算
15
>>>'a'*10                 #字符串重复运算
'aaaaaaaaaa'
```

%运算符表示对整数和浮点数的取模运算。由于受浮点数精确度的影响，计算结果会有误差。例如：

```
>>>5%3
2
>>>-5%3
1
>>>5%-3
-1
>>>10.5%2.1               #浮点数取模运算
2.0999999999999996        #结果有误差
```

**运算符实现乘方运算，其优先级高于*和/。例如：

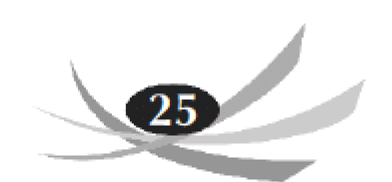
```
>>>2**3
8
>>>3.5**2
12.25
>>>2**3.5
11.313708498984761
>>>4*3**2
36
```

2. 算术表达式

用算术运算符将运算对象(操作数)连接起来且符合Python语言语法规则的式子，称为算术表达式。运算对象包括常量、变量和函数等。例如：

```
3+a*b/5-2.3+'b'
```

就是一个算术表达式。该表达式是先求a*b，然后让其结果再除以5。之后再从左至右计算加法和减法运算。如果表达式中有括号，则应该先计算括号内的运算，再计算括号外的运算。



3. 数据转换

计算机中的运算与数据类型有密切关系。在 Python 中，同一个表达式允许不同类型的数据参加运算，这就要求在运算之前，先将这些不同类型的数据转换成同一类型，然后再进行运算。例如，计算表达式

```
10/4* 4
```

作为数学式子，结果很明显是 10，但在不同的程序设计语言中计算结果就不同了。

在 C 语言中，由于 C 语言遵循“两个整数计算，其结果仍为整数”的原则，先计算 10/4，得到 2，再用 2 乘以 4，得到结果 8。

在 Python 中，将进行除法运算的操作数自动转换成浮点型 10.0/4.0，再进行运算，得到 2.5，再用 2.5 乘以 4，得到结果 10.0。

【例 2.4】 自动类型转换。

```
>>> 10/4* 4
10.0
>>> type(10/4* 4)
<class 'float'>
>>> 10//4* 4
8
>>> type(10//4* +4)
<class 'int'>
```

当自动类型转换达不到转换需求时，可以使用类型转换函数，将数据从一种类型强制(或称为显式)转换成另一种类型，以满足运算需求。常用的类型转换函数如表 2.3 所示。

表 2.3　常用的类型转换函数

函　数	功 能 描 述
int(x)	将 x 转换为整数
float(x)	将 x 转换为浮点数
complex(x)	将 x 转换为复数，其中实部为 x，虚部为 0
complex(x,y)	将 x、y 转换为复数，其中实部为 x，虚部为 y
str(x)	将 x 转换为字符串
chr(x)	将一个整数转换为一个字符，整数为字符的 ASCII 编码
ord(x)	将一个字符转换为它的 ASCII 编码的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串
eval(str)	将字符串 str 当作有效表达式求值，并返回计算结果

【例 2.5】 强制类型转换。

```
>>> x= int("5")+15              #将字符串"5"强制转换为整数
>>> x
20
>>> y= float("5")               #将字符串"5"强制转换为浮点数
>>> y
5.0
>>> complex(x)                  #创建实部为 x,虚部为 0 的复数
(20+0j)
>>> complex(x,y)                #创建实部为 x,虚部为 y 的复数
(20+5j)
>>> z= str(x)                   #读取 x 的值转换为字符串,x 中的值不变
>>> z
'20'
>>> type(x)                     #输出 x 的类型
<class 'int'>                   #x 的类型没有变化
>>> chr(97)                     #得到整数 97 所表示的字符
'a'
>>> ord('A')                    #得到字符'A'的 ASCII 码值
65
>>> hex(x)                      #读取 x 的值转换为十六进制字符串,x 中的值不变
'0x14'
>>> oct(x)                      #读取 x 的值转换为八进制字符串,x 中的值不变
'0o24'
>>> eval('x-y')                 #计算字符串'x-y'表示的表达式的值
15.0
```

2.3.3 赋值运算符和赋值表达式

赋值运算符构成了 Python 语言最基本、最常用的赋值语句,同时 Python 语言还允许赋值运算符与其他的 12 种运算符结合使用,形成复合的赋值运算符,使 Python 语言编写的程序更简单、精练。

1. 赋值运算符

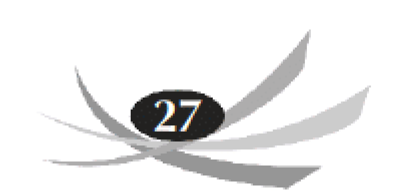
赋值运算符用“=”表示,它的作用是将一个数据赋给一个变量。

例如,“a=3”的作用是把常量 3 赋给变量 a。也可以将一个表达式赋给一个变量,例如,“a=x%y”的作用是将表达式 x%y 的结果赋给变量 a。

赋值运算符“=”是一个双目运算符,其结合方向为从右至左。

2. 赋值表达式

由赋值运算符“=”将一个变量和一个表达式连接起来的式子称为赋值表达式,其一般形式为:



变量=表达式

等号的左边必须是变量，右边是表达式。对赋值表达式的求解过程为：计算赋值运算符右边“表达式”的值，并将计算结果赋给其左边的“变量”。例如：

```
>>> y=2
>>> x=(y+2)/3
>>> x
1.3333333333333333
```

赋值时先计算表达式的值，然后使该变量指向该数据对象，该变量可以理解为该数据对象的别名。

注意：Python 的赋值和一般的高级语言的赋值有很大的不同，它是引用赋值。看下面的代码：

```
>>> a=5
>>> b=8
>>> a=b
```

执行 a＝5 和 b＝8 之后，a 指向的是 5，b 指向的是 8，当执行 a＝b 的时候，b 把自己指向的地址（也就是 8 的内存地址）赋给了 a，那么最后的结果就是 a 和 b 同时指向了 8，如图 2.3 所示。

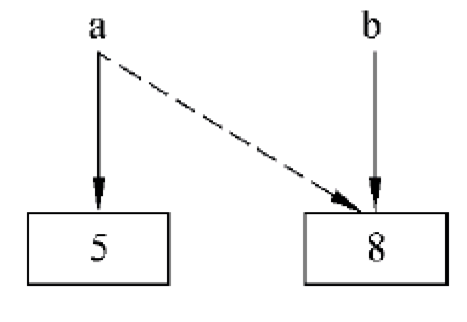


图 2.3　引用赋值

3. 多变量赋值

1）链式赋值

在 Python 中，可通过链式赋值将同一个值赋给多个变量，一般形式为：

```
>>> x=y=5
>>> x
5
>>> y
5
```

这里 x＝y＝5 等价于先执行“y＝5”，再执行“x＝y”。

例如，a＝b＝c＝1，创建了一个整型对象，值为 1，三个变量 a、b、c 被分配到相同的内存空间上，均指向数据对象 1。

2）多变量并行赋值

Python 可以对多个变量并行赋值，一般形式为：

变量 1,变量 2,…,变量 n= 表达式 1,表达式 2, …,表达式 n

变量个数要与表达式的个数一致，其过程为：首先计算表达式右边 n 个表达式的值，然后同时将表达式的值赋给左边的 n 个变量。例如：

```
>>> x,y,z=2,5,8
>>> x
2
```

```
>>>y
5
>>>z
8
```

再看一个特殊的例子,执行结果中变量 x 的值是多少呢?

```
>>>x,x=-10,20
>>>x
20
```

从变量 x 的输出结果 20 可以得知:表达式"x,x=-10,20"先执行 x=-10,后执行 x=20,因此 x 最终的值是 20。

例如:

```
>>>x=20
>>>x,x=3,x*3
>>>x
60
```

首先执行 x=20,x 的值为 20,接着执行语句"x,x=3,x*3",此时先执行 x=3,接着执行 x=x*3,但这时 x 的值是 20,表明是并行赋值,因此最后 x 的值是 60。

采取并行赋值,可以使用一条语句就可以交换两个变量的值:x,y=y,x。

4. 复合的赋值运算符

Python 语言规定,赋值运算符"="与 7 种算术运算符(+、-、*、/、//、**、%)和 5 种位运算符(>>、<<、&、^、|)结合构成 12 种复合的赋值运算符。它们分别是:+=、-=、*=、/=、//=、**=、%=、>>=、>>=、&=、^=和|=,结合方向为自右至左。

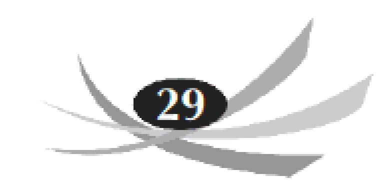
例如:

```
a+=3          等价于     a=a+3
a*=a+3        等价于     a=a*(a+3)
a%=3          等价于     a=a%3
```

注意:"a*=a+3"与"a=a*a+3"是不等价的,"a*=a+3"等价于"a=a*(a+3)",这里括号是必需的。

【例 2.6】 复合的赋值运算符示例。

```
>>>a=3
>>>b=5
>>>a+=b
>>>a
8
>>>a>>=2
>>>a
```




```
2
>>>a*=a+3
>>>a
10
```

2.3.4　关系运算符和关系表达式

1. 关系运算符

关系运算是用来比较关系运算符左右两边的表达式，若比较结果符合给定的条件，则结果是 True（代表真），否则结果是 False（代表假）。Python 语言提供了 7 种关系运算符供程序设计时使用，其属性如表 2.4 所示。

表 2.4　关系运算符属性

运算符	含　义	优　先　级	结　合　性
>	大于	这些运算符的优先级相同，但比下面的运算符优先级低	左结合
>=	大于等于		
<	小于		
<=	小于等于		
==	等于	这些运算符的优先级相同，但比上面的运算符优先级高	
!=	不等于		
<>	不等于		

关系运算符的优先级：{>、>=、<、<=}→{==、!=、<>}。

前 4 个运算符的优先级相同，后 3 个运算符的优先级相同；前 4 个运算符的优先级低于后 3 个运算符的优先级。

2. 关系表达式

由关系运算符和操作数组成的表达式称为关系表达式。关系表达式的运算结果是一个逻辑值，即只有 0 或 1 两个值。在 Python 中，真用 True 表示，假用 False 表示。

例如：

```
>>>x,y,z=2,3,5
>>>x>y
False
>>>y<z
True
>>>x<y<z
True
```

注意浮点数的相等。在计算机中，浮点数是实数的近似值。执行一系列浮点数的运算后，可能会发生四舍五入的情况。例如：

```
>>> x=123456
>>> y=-111111
>>> z=1.2345678
>>> a=(x+y)+z
>>> b=x+(y+z)
>>> a
12346.2345678
>>> b
12346.234567799998
```

在数学中，x、y、z 初始值相同的情况下，(x＋y)＋z 和 x＋(y＋z)结果相同。在计算机中，需要进行四舍五入，因此得到了不同的值。

```
>>> a==b
False
>>> a-b<0.0000001
True
```

对于语句 a＝＝b，目的是检查 a 和 b 是否具有相同的值。a－b＜0.0000001 是检查 a 和 b 是否足够接近。在比较浮点数是否相等时，前一种方法常常会得到不正确的结果，因此一般都采用后一种方法。

Python 中，关系运算符可以连用，也称关系运算符链，其计算方法与数学中的计算方法相同。例如：

```
>>> x=5
>>> 0<=x<=10          #x大于等于 0且小于等于 10
True                  #表达式结果为真
>>> 0<=x<=3           #x大于等于 0且小于等于 3
False                 #表达式结果为假
```

2.3.5 逻辑运算符和逻辑表达式

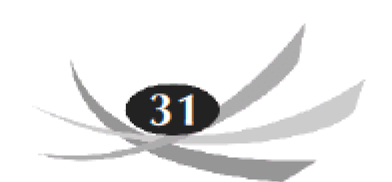
1. 逻辑运算符

逻辑运算符是对关系表达式或逻辑值进行运算的运算符，运算结果仍是逻辑值。Python 语言提供三种逻辑运算符，其属性如表 2.5 所示。

表 2.5 逻辑运算符属性

运算符	含义	优先级	结合性
not	逻辑非	↑高	右结合
and	逻辑与		左结合
or	逻辑或	低	

and 和 or 是双目运算符，结合方向是自左至右，且 and 的优先级高于 or。not 是单目



运算符,结合方向是自右至左,它的优先级高于前两种。

三种逻辑运算符的意义分别如下。

(1) a and b:若 a 和 b 两个运算对象同时为真,则结果为真,否则只要有一个为假,结果就为假。例如:

```
15>13 and 14>12
```

由于 15>13 为真,14>12 也为真,逻辑与的结果为真值 True。

(2) a or b:若 a 和 b 两个运算对象同时为假,则结果为假,否则只要有一个为真,结果就为真。例如:

```
15<10 or 15<118
```

由于 15<10 为假,15<118 为真,逻辑或的结果为真值 False。

(3) not a:若 a 为真时,结果为假;反之,若 a 为假,结果为真。例如:not (15>10)的结果为假值 False。

三种逻辑运算符的真值表如表 2.6 所示。

表 2.6　逻辑运算符真值表

a	b	a and b	a or b	not a
真	真	真	真	假
真	假	假	真	假
假	真	假	真	真
假	假	假	假	真

2. 逻辑表达式

由逻辑运算符连接关系表达式或逻辑值组成的表达式称为逻辑表达式。逻辑表达式的运算结果取最后计算的那个表达式的值。

在逻辑表达式的求解中,并不是所有的逻辑运算符都要被执行,只有在必须执行下一个逻辑运算符才能求出表达式的解时,才执行该运算符。其运算规则如下。

(1) 对于与运算 a and b。

如果 a 为真,继续计算 b,b 将决定最终整个表达式的真值,所以,结果为 b 的值。

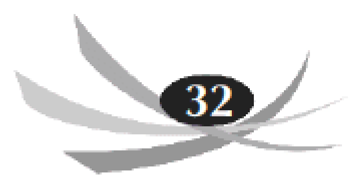
如果 a 为假,无须计算 b,就可以得知整个表达式的真值为假,所以,结果为 a 的值。

例如:

```
>>> True and 0
```




```
>>> False and 12
False
```





```
>>> True and 12 and 0
0
```

(2) 对于或运算 a or b。

如果 a 为真，无须计算 b，就可得知整个表达式的真值为真，所以结果为 a 的值。

如果 a 为假，继续计算 b，b 将决定整个表达式最终的值，所以结果为 b 的值。

例如：

```
>>> True or 0
True
>>> False or 12
12
>>> False or 12 or 0
12
```

2.3.6 成员运算符和成员表达式

成员运算符用于判断一个元素是否在某一个序列中，或者判断一个字符是否属于这个字符串等，运算结果仍是逻辑值。Python 提供了两种成员运算符，其属性如表 2.7 所示。

表 2.7 成员运算符

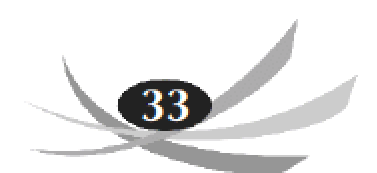
运算符	含 义	优 先 级	结 合 性
in	存在	相同	左结合
not in	不存在		

in 运算符用于在指定的序列中查找某个值是否存在，存在则返回 True，不存在则返回 False。例如：

```
>>> 'a' in 'abcd'
True
>>> 'ac' in 'abcd'
False
```

not in 运算符用于在指定的序列中查找某个值是否不存在，不存在则返回 True，存在则返回 False。例如：

```
>>> 'a' not in 'bcd'
True
>>> 3 not in [1,2,3,4]
False
```



2.3.7　同一性运算符和同一性表达式

同一性运算符用于测试两个变量是否指向同一个对象，其运算结果是逻辑值。Python 提供两种同一性运算符，其属性如表 2.8 所示。

表 2.8　同一性运算符属性

运算符	含　义	优 先 级	结 合 性
is	相同	相同	左结合
is not	不相同		

is 用来检查运算的两个变量是否引用同一对象，也就是 id 是否相同，如果相同则返回 True，不相同则返回 False。例如：

```
>>> x= y= 2.5
>>> z= 2.5
>>> x is y
True
>>> x is z
False
```

在该例中，变量 x 和 y 被绑定到同一个整数上，而 z 被绑定到另一个与 x 值具有相同数值的另一个对象上，也就是 x 和 z 值相等，但不是同一个对象。

is not 用来检查运算的两个变量是否不是引用同一对象，如果不是同一个对象则返回 True，否则返回 False。

```
>>> x is not z
True
```

注意区分 is 与==：

```
>>> x= y= 2.5
>>> z= 2.5
>>> x= = z
True
>>> x is z
False
>>> print(id(x))
45298176
>>> print(id(y))
45298176
```

```
>>>print(id(z))
44866880
```

从运行结果可以看出，x 和 y 的 id 相同；x 和 z 的值相等，但 id 不同。

2.4 运算符的优先级和结合性

Python 中不同的运算符具有不同的优先级。在计算表达式的值时，先计算优先级比较高的运算符，如果表达式中的运算符优先级相同，还要按照运算符的结合性确定计算的先后次序。当然可以使用圆括号改变运算的顺序。表 2.9 列出了本章所介绍的运算符的优先级和结合性。

表 2.9 常用运算符的优先级和结合性

优先级	运算符	结合性
高	()	从左至右
↑	**	
	*、/、%、//	
	+、-	
	<、<=、>、>=	
	==、!=、<>	
	not	从右至左
	and	从左至右
	or	
	is、not is	
低	in、not in	
	=、+=、-=、*=、/=、%=、//=、**==	从右至左

习　题

1. 选择题

(1) 表达式 16/4－2**5*8/4%5//2 的值为(　　)。

A. 14　　B. 4　　C. 20　　D. 2

(2) 数学关系表达式 $3\leqslant x\leqslant 10$ 表示成正确的 Python 表达式为(　　)。

A. 3<=x<10　　B. 3<=x and x<10

C. x>=3 or x<10　　D. 3<=x and<=10

(3) 以下不合法的表达式是(　　)。

A. x in [1,2,3,4,5]　　B. x－6＞5

C. e＞5 and 4＝＝f　　D. 3＝a

(4) Python 语句 print(0xA＋0xB)的输出结果是(　　)。

A. 0xA＋0xB　　B. A＋B　　C. 0xA＋0xB　　D. 21

(5) 下列表达式中,值不是 1 的是(　　)。

A. 4//3　　B. 15%2　　C. 1^0　　D. ～1

(6) 语句 eval('2＋4/5')执行后的输出结果是(　　)。

A. 2.8　　B. 2　　C. 2＋4/5　　D. '2＋4/5'

(7) 若字符串 s＝'a\nb\tc',则 len(s)的值是(　　)。

A. 7　　B. 6　　C. 5　　D. 4

(8) 下列表达式的值为 True 的是(　　)。

A. 2!＝5 or 0　　B. 3＞2＞2　　C. 5＋4j＞2－3j　　D. 1 and 5＝＝0

(9) 与关系表达式 x＝＝0 等价的表达式是(　　)。

A. x＝0　　B. not x　　C. x　　D. x!＝1

2. 填空题

(1) n 是小于正整数 k 的偶数,用 Python 表达式表示为________。

(2) 若 a＝7,b＝－2,c＝4,则表达式 a%3＋b＊b－c/5 的值为________。

(3) Python 表达式 1/2 的值为________,1//3＋1/3＋1%3 的值为________。

(4) 计算 $2^{31}-1$ 的 Python 表达式是________。

(5) 数学表达式 $\frac{e^{|x-y|}}{3^x+\sqrt{6}\sin y}$ 的 Python 表达式为________。

(6) 已知“a＝3;b＝5;c＝6;d＝True”,则表达式 not d or a＞＝0 and a＋c＞b＋3 的值是________。

(7) Python 语句“x＝0;y＝True;print(x＞y and 'A'＜'B')”的运行结果是________。

(8) 判断整数 i 能否被 3 和 5 整除的 Python 表达式为________。

3. 写出下面各逻辑表达式的值,其中 a＝3,b＝4,c＝5。

(1) a＋b＞c and b＝＝c。

(2) a or b＋c and b＞c。

(3) not(a＞b) and not c or 1。

(4) not(a＋b)＋c and b＋c/2。

第 3 章 顺序程序设计

程序由多条语句构成，它描述计算机的执行步骤。人们利用计算机解决问题，必须预先将问题转化为计算机语句描述的解题步骤，即程序。也就是说，程序在计算机上执行时，程序中的语句完成具体的操作并控制计算机的执行流程，但程序并不一定完全按照语句序列的书写顺序来执行。程序中语句的执行顺序称为程序结构。程序包含三种基本结构：顺序结构、选择结构和循环结构。如果程序中的语句是按照书写顺序执行，则称其为顺序结构；如果程序中某些语句按照某个条件来决定是否执行，则称其为选择结构；如果程序中某些语句反复执行多次，则称其为循环结构。

顺序结构是最简单的一种结构，它只需按照处理顺序依次写出相应的语句即可。因此，学习程序设计，首先从顺序结构开始。本章主要介绍算法的概念、程序的基本结构、数据的输入与输出及顺序程序设计方法。

3.1 算　　法

开发程序的目的，就是要解决实际问题。然而，面对各种复杂的实际问题，如何编写程序，往往令初学者感到茫然。程序设计语言只是一种工具，只懂得语言的规则并不能保证编写出高质量的程序。程序设计的关键是设计算法，算法与程序设计和数据结构密切相关。简单地讲，算法是解决问题的策略、规则和方法。算法的具体描述形式很多，但计算机程序是对算法的一种精确描述，而且可在计算机上运行。

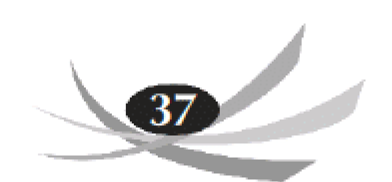
3.1.1 算法的概念

算法就是解决问题的一系列操作步骤的集合。例如，厨师做菜时，要经过一系列的步骤——洗菜、切菜、配菜、炒菜和装盘。用计算机解题的步骤就叫算法，编程人员必须告诉计算机先做什么，再做什么，这可以通过高级语言的语句来实现。通过这些语句，一方面体现了算法的思想，另一方面指示计算机按算法的思想去工作，从而解决实际问题。程序就是由一系列的语句组成的。

著名的计算机科学家沃思(Niklaus Wirth)曾经提出一个著名的公式：

数据结构＋算法＝程序

数据结构是指对数据(操作对象)的描述，即数据的类型和组织形式，算法则是对操作步骤的描述。也就是说，数据描述和操作描述是程序设计的两项主要内容。数据描述的主要内容是基本数据类型的组织和定义，数据操作则是由语句来实现的。算法具有下列特性。



1. 有穷性

任意一组合法输入值，在执行有穷步骤之后一定能结束，即算法中的每个步骤都能在有限时间内完成。

2. 确定性

算法的每一步必须是确切定义的，使算法的执行者或阅读者都能明确其含义及如何执行，并且在任何条件下，算法都只有一条执行路径。

3. 可行性

算法应该是可行的，算法中的所有操作都必须足够基本，都可以通过已经实现的基本操作运算有限次实现。

4. 有输入

一个算法应有零个或多个输入，它们是算法所需的初始量或被加工对象的表示。有些输入量需要在算法执行过程中输入，而有的算法表面上可以没有输入，实际上已被嵌入到算法之中。

5. 有输出

一个算法应有一个或多个输出，它是一组与输入有确定关系的量值，是算法进行信息加工后得到的结果，这种确定关系即为算法的功能。

6. 有效性

在一个算法中，要求每一个步骤都能有效地执行。

以上这些特性是一个正确的算法应具备的特性，在设计算法时应该注意。

3.1.2　算法的评价标准

什么是"好"的算法？通常从下面几个方面来衡量算法的优劣。

1. 正确性

正确性指算法能满足具体问题的要求，即对任何合法的输入，算法都会得出正确的结果。

2. 可读性

可读性指算法被理解的难易程度。算法主要是为了人的阅读与交流，其次才是被计算机执行，因此算法应该更易于人的理解。另外，晦涩难读的程序易于隐藏较多错误而难以调试。

3. 健壮性

健壮性(鲁棒性)即对非法输入的抵抗能力。当输入的数据非法时，算法应当恰当地做出反应或进行相应处理，而不是产生奇怪的输出结果。并且，处理出错的方法不应是中断程序的执行，而应是返回一个表示错误或错误性质的值，以便在更高的抽象层次上进行处理。

4. 高效率与低存储量需求

通常，效率指的是算法的执行时间；存储量指的是算法执行过程中所需的最大存储空

间，两者都与问题的规模有关。尽管计算机的运行速度提高很快，但这种提高无法满足问题规模加大带来的速度要求。所以追求高速算法仍然是必要的。相比起来，人们会更多地关注算法的效率，但这并不是因为计算机的存储空间是海量的，而是由人们面临的问题的本质决定的。二者往往是一对矛盾，常常可以用空间换时间，也可以用时间换空间。

3.1.3 算法的表示

算法就是对特定问题求解步骤的描述，可以说是设计思路的描述。在算法定义中，并没有规定算法的描述方法，所以它的描述方法可以是任意的，既可以用自然语言描述，也可以用数学方法描述，还可以用某种计算机语言描述。若用计算机语言描述，就是计算机程序。

为了能清晰地表示算法，程序设计人员采用更规范的方法。常用的有自然语言描述、流程图、N-S结构流程图和伪代码等。

1. 自然语言描述

自然语言就是人们日常生活中应用的语言。用自然语言表示通俗易懂，容易被人们接受，也更容易学习和表达，但自然语言文字冗长，而且容易产生歧义。假如有这样一句话："他看到我很高兴。"请问这句话表达了他高兴还是我高兴？仅从这句话本身很难判断。此外，用自然语言描述包含分支和循环的算法十分不方便。因此，除了一些十分简单的算法外，一般不采用自然语言来描述算法。

2. 流程图

流程图是描述算法最常用的一种方法，利用集合图形符号来代表不同性质的操作，用流程线来知识算法的执行方向。ANSI(美国国家标准化协会)规定的一些常用流程图符号如图3.1所示。这种表示直观、灵活，很多程序员采用这种表示方法，因此又称传统的流程图。本书中的算法将采用这种表示方法描述，读者应熟练掌握这种流程图。

【例3.1】 设计一个算法，求三个整数的和，画出流程图。

求三个整数和的算法流程图如图3.2所示。

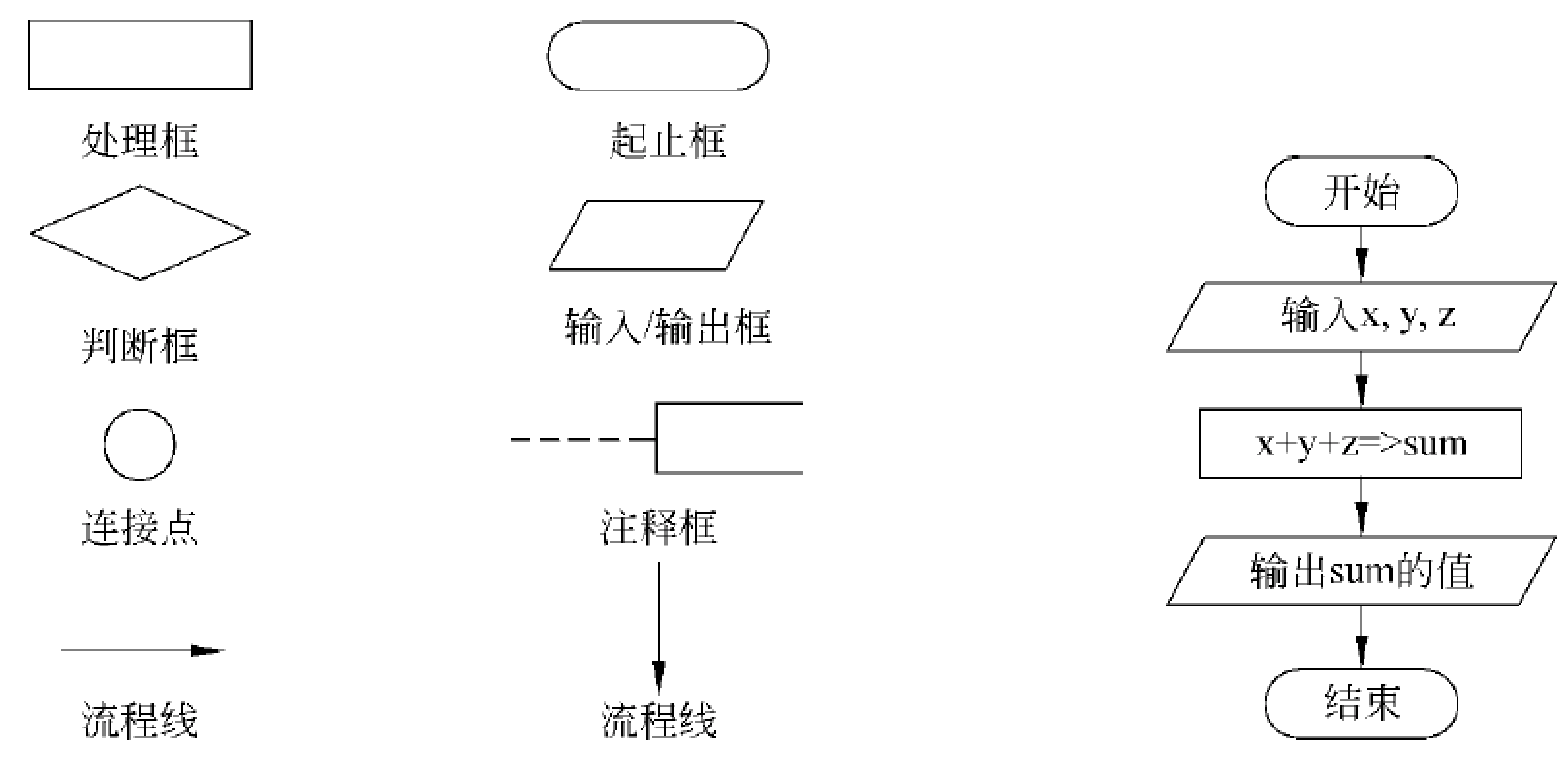


图3.1 常用流程图符号　　**图3.2 求三个整数和的算法流程图**

注意： 画流程图时，每个框内要说明操作内容，描述要确切，不要有二义性。画箭头时应注意箭头的方向，箭头方向表示程序执行的流向。

【例 3.2】 求两个正整数的最大公约数。

求两个正整数的最大公约数的算法流程图如图 3.3 所示。

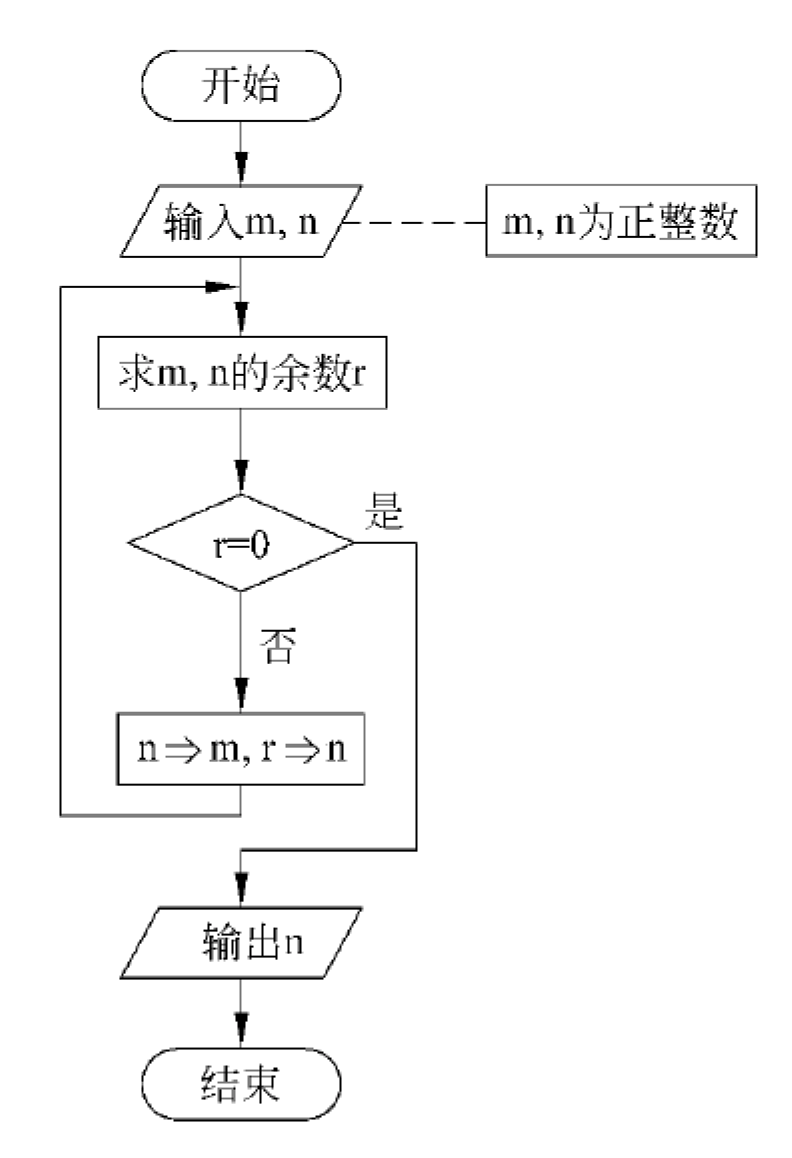


图 3.3　求两个正整数的最大公约数的算法流程图

【例 3.3】 设计解一元二次方程 $ax^2+bx+c=0(a\neq0)$的算法，画出流程图。

分析：求解一元二次方程可按照以下步骤完成。

(1) 计算 $\Delta=b^2-4ac$。

(2) 如果 $\Delta<0$，则原方程无实数解。否则($\Delta\geqslant0$)，计算：

$$x_1=\frac{-b+\sqrt{b^2-4ac}}{2a}\ ;\quad x_2=\frac{-b-\sqrt{b^2-4ac}}{2a}$$

(3) 输出解 x_1、x_2 或无实数解信息。

流程图如图 3.4 所示。

3. N-S 结构流程图

N-S 结构流程图是美国学者 I. Nassi 和 B. Shneiderman 于 1973 年提出的一种新的流程图形式。在这种流程图中完全去掉了流程线，全部算法写在一个矩形框内，而且在框内还可以包含其他框。这样算法被迫只能从上到下顺序执行，从而避免了算法流程的任意转向，保证了程序的质量。

例 3.1 的 N-S 结构流程图如图 3.5 所示。

例 3.2 的 N-S 结构流程图如图 3.6 所示。

4. 伪代码

伪代码是介于自然语言和计算机语言之间的文字和符号，是帮助程序员指定算法的智能化语言，它不能在计算机上运行，但使用起来比较灵活，无固定格式规范，只要写出来自己或别人能看懂即可。由于它与计算机语言比较接近，因此易于转换为计算机程序。

```
input a,b,c
Δ=b*b—4ac
if    Δ<0    then
print"方程无实数解"
else
x1=(-b+sqrt(Δ))/(2*a),
x2=(-b-sqrt(Δ))/(2*a)
print  x1,x2
```

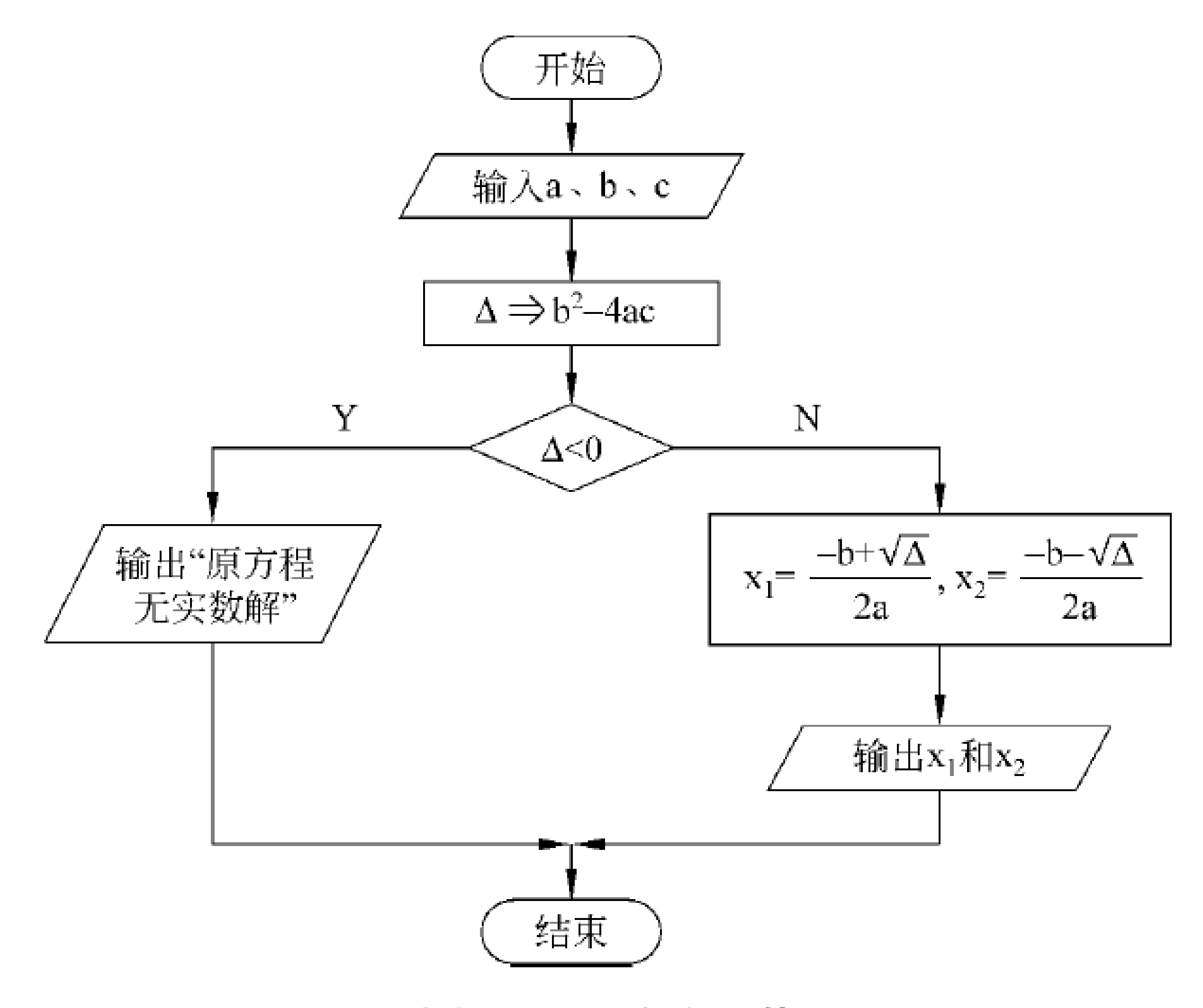


图 3.4　求解一元二次方程算法流程图



输入x,y,z
sum =x+y+z
输出sum的值

图 3.5　例 3.1 的 N-S 结构流程图

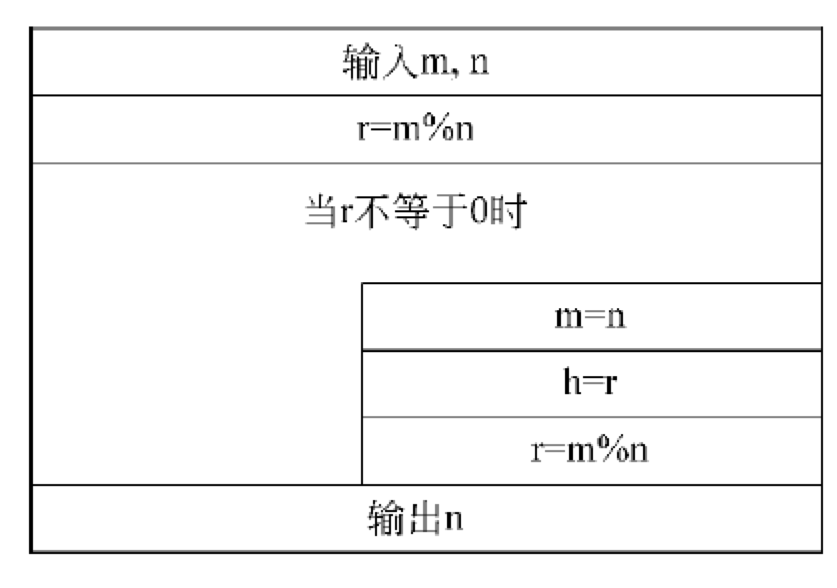


图 3.6　例 3.2 的 N-S 结构流程图

在以上几种描述算法的方法中，具有熟练编程经验的人士喜欢用伪代码，初学者使用流程图或 N-S 结构流程图较多，易于理解，比较形象。

3.2　程序的基本结构

随着计算机的发展，编写的程序越来越复杂。一个复杂程序多达数千万条语句，而且程序的流向也很复杂，常常用无条件转向语句去实现复杂的逻辑判断功能。因而造成程序质量差，可靠性很难保证，同时也不易阅读，维护困难。20 世纪 60 年代末期，国际上出

现了所谓的软件危机。

为了解决这一问题，就出现了结构化程序设计，它的基本思想是像玩积木游戏那样，只要有几种简单类型的结构，可以构成任意复杂的程序。这样可以使程序设计规范化，便于用工程的方法来进行软件生产。基于这样的思想，1966 年意大利的 Bobra 和 Jacopini 提出了三种基本结构，即顺序结构、选择结构和循环结构，由这三种基本结构组成的程序就是结构化程序。

3.2.1　顺序结构

顺序结构是最简单的一种结构，其语句是按书写顺序执行的，除非指示转移，否则计算机自动以语句编写的顺序一句一句地执行。顺序结构的语句程序流向是沿着一个方向进行，有一个入口(A)和一个出口(B)。流程图和 N-S 结构流程图如图 3.7 和图 3.8 所示，先执行程序模块 A，然后再执行程序模块 B，程序模块 A 和程序模块 B 分别代表某些操作。

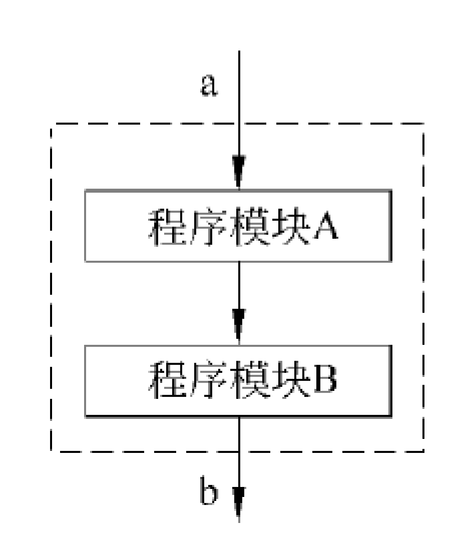


图 3.7　顺序结构的流程图表示

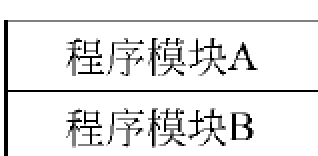


图 3.8　顺序结构的 N-S 结构流程图表示

3.2.2　选择结构

在选择结构中，程序可以根据某个条件是否成立，选择执行不同的语句。选择结构如图 3.9 和图 3.10 所示。当条件成立时执行程序模块 A，否则执行程序模块 B。程序模块 B 也可以为空，如图 3.11 所示。当条件为真时执行某个指定的操作(程序模块 A)，条件为假时跳过该操作(单路选择)。

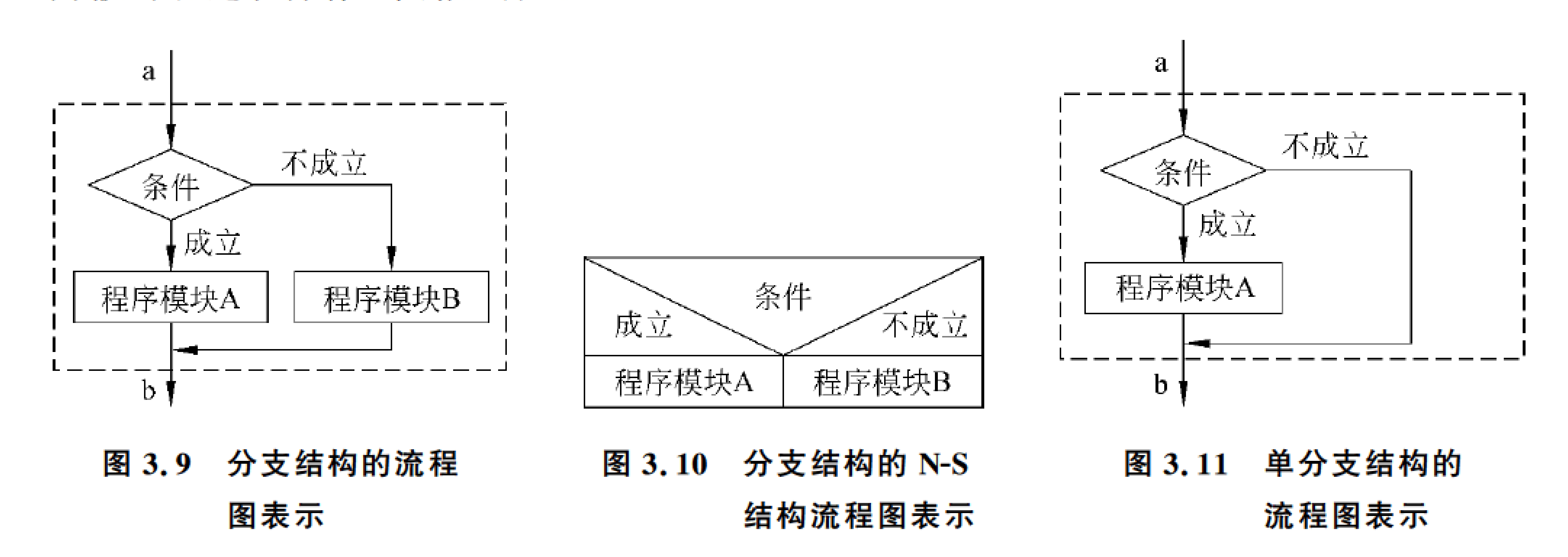


图 3.9　分支结构的流程图表示

图 3.10　分支结构的 N-S 结构流程图表示

图 3.11　单分支结构的流程图表示

3.2.3　循环结构

在循环结构中，可以使程序根据某种条件和指定的次数，使某些语句执行多次。循环

结构有两种形式：当型循环和直到型循环。

1. 当型循环

先判断，只要条件成立(为真)就反复执行程序模块；当条件不成立(为假)时则结束循环。当型循环结构的流程图和N-S结构流程图分别如图3.12(a)和图3.12(b)所示。

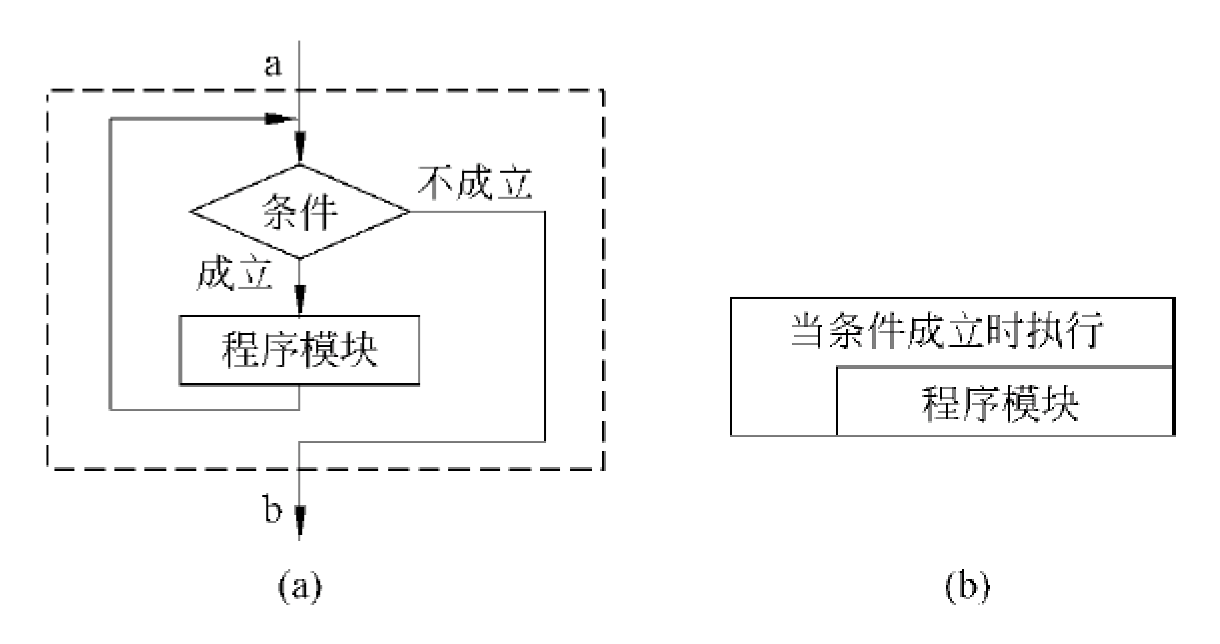


图3.12 当型循环结构的流程图和N-S结构流程图

2. 直到型循环

先执行程序模块，再判断条件是否成立。如果条件成立(为真)则继续执行循环体；当条件不成立(为假)时则结束循环。直到型循环结构的流程图和N-S结构流程图如图3.13所示。

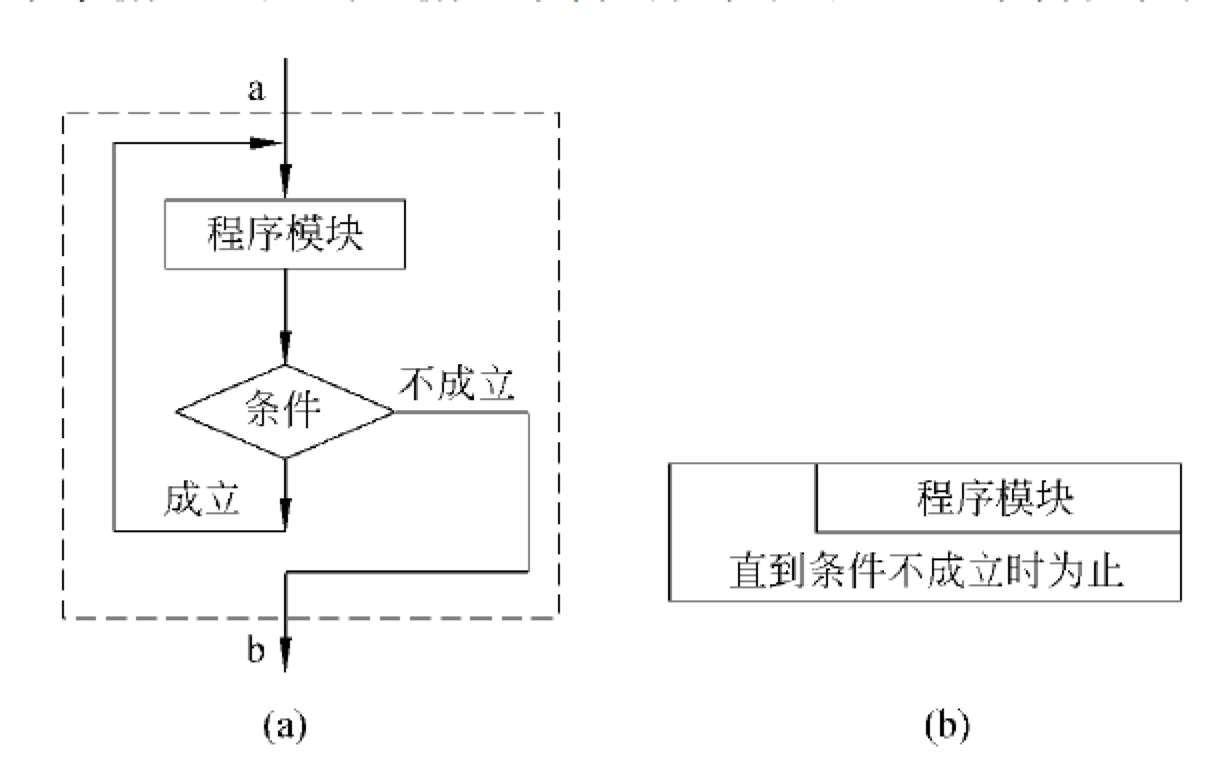
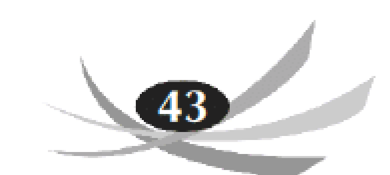


图3.13 直到型循环结构的流程图和N-S结构流程图

注意：无论是顺序结构、选择结构还是循环结构，它们都有一个共同的特点，即只有一个入口和一个出口。从示意的流程图可以看到，如果把基本结构看作一个整体(用虚线框表示)，执行流程从a点进入基本结构，而从b点脱离基本结构。整个程序由若干个这样的基本结构组成。三种结构之间可以是平行关系，也可以相互嵌套，通过结构之间的复合形成复杂的结构。结构化程序的特点就是单入口、单出口。

3.3 数据的输入与输出

通常，一个程序可以分成三步进行：输入原始数据、进行计算处理和输出运行结果。其中，数据的输入与输出是用户通过程序与计算机进行交互的操作，是程序的重要组成部



分。本节详细介绍 Python 的输入与输出。

3.3.1　标准输入与输出

1. 标准输入

Python 提供了内置函数 input()从标准输入设备读入一行文本，默认的标准输入设备是键盘。input()函数的基本格式为：

```
input([提示字符串])
```

说明：方括号中的提示字符串是可选项，如果有"提示字符串"，运行时原样显示，给用户以提示。

在 Python 2.x 和 Python 3.x 中该函数的使用方法略有不同。

在 Python 2.x 中，该函数返回结果的类型由输入时所使用的界定符来决定。例如：

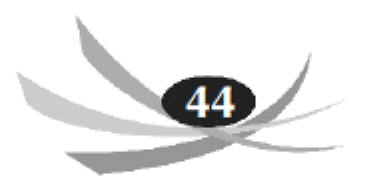
```
>>> x= input("Please enter your input: ")
Please enter your input: 5              #没有界定符,x为整数
>>> x= input("Please enter your input: ")
Please enter your input: '5'            #单引号界定符,x为字符串
>>> x= input("Please enter your input: ")
Please enter your input: [1,2,3]        #方括号界定符,x为列表
>>> x= input("Please enter your input: ")
Please enter your input: (1,2,3)        #圆括号界定符,x为元组
```

在 Python 2.x 中还提供一个内置函数 raw_input()用来接收用户输入的值，该函数将所有用户的输入都作为字符串看待，返回字符串类型。例如：

```
>>> x= raw_input ("Please enter your input: ")
Please enter your input: 5
>>> x
'5'
>>> x= raw_input ("Please enter your input: ")
Please enter your input: (1,2,3)
>>> x
'(1,2,3)'
```

在 Python 3.x 中，将 raw_input()和 input()进行了整合，去除了 raw_input()函数，仅保留了 input()函数。input()函数接收任意任性输入，将所有输入默认为字符串处理，并返回字符串类型，相当于 Python 2.x 中的 raw_input()函数。例如：

```
>>> x= input("Please enter your input: ")
Please enter your input: 5
>>> print(type(x))
```




```
<class'str'>
```

说明：内置函数 type()用来返回变量类型。当输入数值 5 赋值给变量 x 之后，x 的类型为字符串类型。

```
>>> x= input ("Please enter your input:")
Please enter your input: (1,2,3)
>>> print(type(x))
<class'str'>
```

如果要输入数值类型数据，可以使用类型转换函数将字符串转换为数值。例如：

```
>>> x= int(input ("Please enter your input:"))
Please enter your input:5
>>> print(type(x))
<class 'int'>
```

说明：x 接收的是字符串 5，通过 int()函数将字符串转换为整型类型。

input()函数也可给多个变量赋值。例如：

```
>>> x,y= input()
3,4
>>> x
3
>>> y
4
```

2. 标准输出

在 Python 2.x 和 Python 3.x 中输出方法也不完全一致。在 Python 2.x 中使用 print 语句进行输出，在 Python 3.x 中使用 print()函数进行输出。

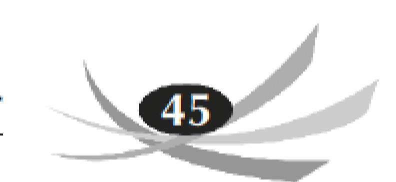
本书给出的例子大部分是在 Python 3.5.3 环境下编写运行，因此这里重点介绍 print()函数的用法。

print()函数一般形式为：

```
print([输出项 1,输出项 2,…,输出项 n][,sep=分隔符][,end=结束符])
```

说明：输出项之间用逗号分隔，没有输出项时输出一个空行。sep 表示输出时各输出项之间的分隔符(默认以空格分隔)，end 表示输出时的结束符(默认以回车换行结束)。print()函数从左求出至右各输出项的值，并将各输出项的值依次显示在屏幕的同一行上。例如：

```
>>> x,y=2,3
>>> print(x,y)
2 3
```




```
>>>print(x,y,sep=':')
2:3
>>>print(x,y,sep=':',end='%')
2:3%
```

3.3.2 格式化输出

在很多实际应用中都需要将数据按照一定格式输出。

1. 字符串格式化%

Python 中 print()函数可以按照指定的输出格式在屏幕上输出相应的数据信息。其基本做法是：将输出项格式化，然后利用 print()函数输出。

在 Python 中格式化输出时，采用%分隔格式控制字符串与输出项，一般格式为：

```
格式控制字符串%(输出项 1,输出项 2,…,输出项 n)
```

其功能是按照“格式控制字符串”的要求，将输出项 1，输出项 2，…，输出项 n 的值输出到输出设备上。

其中，格式控制字符串用于指定输出格式，它包含如下两类字符。

(1) 常规字符：包括可显示的字符和用转义字符表示的字符。

(2) 格式控制符：以%开头的一个或多个字符，以说明输出数据的类型、形式、长度、小数位数等，如“%d”表示按十进制整型输出；“%c”表示按字符型输出等。格式控制符与输出项应一一对应。

对应不同类型数据的输出，Python 采用不同的格式说明符描述。格式说明详见表 3.1。

表 3.1　print()的格式说明

格式符	格 式 说 明
d 或 i	以带符号的十进制整数形式输出整数(正数省略符号)
o	以八进制无符号整数形式输出整数(不输出前导 0)
x 或 X	以十六进制无符号整数形式输出整数(不输出前导 0x)。用 x 时，以小写形式输出包含 a、b、c、d、e、f 的十六进制数；用 X 时，以大写形式输出包含 A、B、C、D、E、F 的十六进制数
c	以字符形式输出，输出一个字符
s	以字符串形式输出
f	以小数形式输出实数，默认输出 6 位小数
e 或 E	以标准指数形式输出实数，数字部分隐含 1 位整数，6 位小数。使用 e 时，指数以小写 e 表示，使用 E 时，指数以大写 E 表示
g 或 G	根据给定的值和精度，自动选择 f 与 e 中较紧凑的一种格式，不输出无意义的 0

例如：

```
print("sum=%d"%x)
```

若 x=300，则输出为：

```
sum= 300
```

格式控制字符串中“sum=”照原样输出,“%d”表示以十进制整数形式输出。

2. 附加格式说明符

对输出格式,Python 语言同样提供附加格式说明符,用以对输出格式做进一步描述。在使用表 3.1 所示的格式控制字符时,在%和格式字符之间可以根据需要使用下面的几种附加格式说明符,使得输出格式的控制更加准确。附加格式说明符详见表 3.2。

表 3.2 附加格式说明符

附加格式说明符	格式说明
m	域宽,十进制整数,用以描述输出数据所占宽度。如果 m 大于数据实际位数,输出时前面补足空格;如果 m 小于数据的实际位数,按实际位数输出。当为小数时,小数点或占 1 位
n	附加域宽,十进制整数,用于指定实型数据小数部分的输出位数。如果 n 大于小数部分的实际位数,输出时小数部分用 0 补足;如果 n 小于小数部分的实际位数,输出时将小数部分多余的位四舍五入。如果用于字串数据,表示从字串中截取的字符数
-	输出数据左对齐,默认时为右对齐
+	输出正数时,也以+号开头
#	作为 o,x 的前缀时,输出结果前面加上前导符号 0、0x

这样,格式控制字符的形式为:

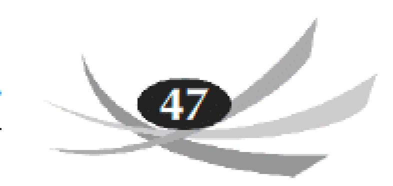
```
%[附加格式说明符]格式符
```

注意:书中对语句格式进行描述时用方括号表示可选项,其余出现在格式中的非汉字字符均为定义符,应原样照写。

例如,可在%和格式字符之间加入形如“m.n”(m,n 均为整数,含义见表 3.2)的修饰。其中,m 为宽度修饰,n 为精度修饰。如%7.2f,表示用实型格式输出,附加格式说明符“7.2”表示输出宽度为 7,输出 2 位小数。

下面是一些格式化输出的实例。

```
>>> year = 2017
>>> month = 1
>>> day = 28
#格式化日期,%02d将数字转换成 2位整型,缺位补 0
>>print('%04d-%02d-%02d'%
(Year,Month,Day))
2017-01-28                    #输出结果
>>> value = 8.123
>>print('%06.2f'%value)       #保留宽度为 6,小数点后 2位小数的数据
```




```
008.12                          #运行结果
>>>print('%d'%10)               #输出十进制数
10
>>>print('%o'%10)               #输出八进制数
12
>>>print('%02x'%10)             #输出 2位十六进制数,字母小写,缺位补 0
0a
>>>print('%04X'%10)             #输出 4位十六进制数,字母大写,缺位补 0
000A
>>>print('%.2e'%1.2888)         #以科学计数法输出浮点型数,保留 2位小数
1.29e+00
```

3.4　顺序程序设计举例

到目前为止,介绍的程序都是逐条语句书写的,程序的执行也是按照顺序逐条执行的,这种程序被称为顺序程序。

下面是能够实现实际功能的顺序程序设计的例子,虽然不难,但对形成清晰的编程思路是有帮助的。

【例 3.4】 从键盘输入一个 3 位整数,分离出它的个位、十位和百位并分别在屏幕输出。

分析:此题要求设计一个从 3 位整数中分离出个位、十位和百位数的算法。例如,输入的数是 235,则输出分别是 2、3、5。百位数字可采用对 100 整除的方法得到,235//100＝2;个位数字可采用对 10 求余的方法得到,235％10＝5;十位数字可通过将其十位数字变化为最高位后再整除的方法得到,(235－2＊100)//10＝3,也可通过将其十位数字变换为最低位再求余的方法得到,(235/10)％10＝3。

根据以上分析,程序应分为 3 步完成。

(1) 调用 input 函数输入该 3 位整数。

(2) 利用上述算法计算该数的个位、十位和百位数。

(3) 输出计算后的结果。

程序如下:

```
x=int(input("请输入一个 3位整数"))
a=x//100
b=(x-a*100)//10
c=x%10
print("百位=%d,十位=%d,个位=%d"%(a,b,c))
```

程序运行结果:

```
请输入一个3位整数 235
百位=2,十位=3,个位=5
```

【例 3.5】 小写字母转盘如图 3.14 所示。用户输入一个小写字母,求出该字母的前驱和后继字母。例如,c 的前驱和后继字母分别是 b 和 d,a 的前驱和后继字母分别是 z 和 b,z 的前驱和后继字母分别是 y 和 a。

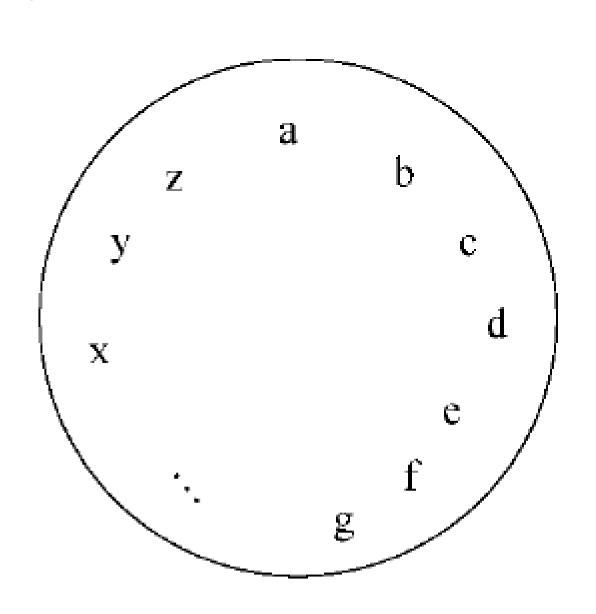


图 3.14 小写字母转盘

分析:首先应该输入一个小写字母存储到字符类型变量(假设为 ch 变量)中,接着再求该字母的前驱和后继字母。

求一个字母的前驱字母并不是简单地减 1,例如,字母 a 的前驱是 z,不能通过减 1 来实现。在没有学习条件控制之前,可以利用取余操作的特性,即任何一个整数除以 26(26 个字母)的余数只能在 0~25。我们可以以 z 为参考点,首先求出输入的字符 ch(假设是 w)与 z 之间的字符偏移数 n='z'-ch='z'-'w'=3,而(n+1)%26=4 则是 ch(字母 w)的前驱字母相对于 z 的偏移数,'z'-(n+1)%26=122-4=118(即字母 v)就是 ch(字母 w)的前驱字母。

采用同样的道理去求后继字母。

程序如下:

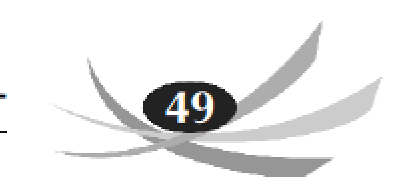
```
ch=input("请输入一个字母:")
pre=ord('z')-(ord('z')-ord(ch)+1)%26          #ord()函数用来得到字母的 ASCII 值
next=ord('a')+(ord(ch)-ord('a')+1)%26
print("%c的前驱字母是%c,后继字母是%c"%(ch,pre,next))
```

程序运行结果:

```
请输入一个字母:z
z的前驱字母是 y,后继字母是 a
```

再次运行程序,结果如下:

```
请输入一个字母:a
a的前驱字母是 z,后继字母是 b
```



习　　题

1. 什么是算法？算法的基本特征是什么？

2. 编写一个加法和乘法计算器程序。

3. 编写程序，输入三角形的 3 条边长 a、b、c，求三角形的面积 area，并画出算法的流程图和 N-S 结构流程图。公式为

$$\text{area} = \sqrt{S(S-a)(S-b)(S-c)}$$

其中，$S=(a+b+c)/2$。

4. 编写程序，输入 4 个数，并求它们的平均值。

5. 从键盘上输入一个大写字母，并将大写字母转换成小写字母并输出。

第4章 选择结构程序设计

选择结构又称分支结构，它根据给定的条件是否满足，决定程序的执行路线。在不同的条件下，执行不同的操作，这在实际求解问题过程中是大量存在的。例如，输入一个整数，要判断它是否为偶数，就需要使用选择结构来实现。根据程序执行路线或分支的不同，选择结构又分为单分支、双分支和多分支三种类型。本章主要介绍 Python 中 if 语句及选择结构程序设计方法。

4.1 单分支选择结构

用 if 语句可以构成选择结构，它根据给定的条件进行判断，以决定执行某个分支程序段。Python 的 if 语句有三种基本形式。

if 语句的一般格式为：

```
if 表达式：
    语句块
```

其语句功能是先计算表达式的值，若为真，则执行语句，否则跳过语句执行 if 语句的下一条语句。其执行过程如图 4.1 所示。

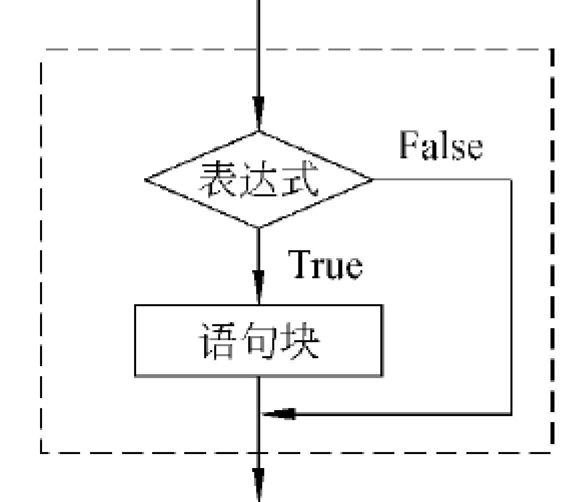


图 4.1 单分支 if 语句的执行过程

注意：

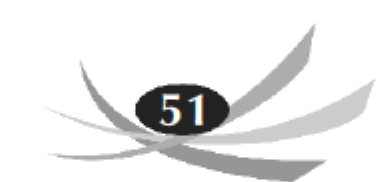
(1) if 语句的表达式后面必须加冒号。

(2) 因为 Python 把非 0 当作真，0 当作假，所以表示条件的表达式不一定必须是结果为 True 或 False 的关系表达式或逻辑表达式，可以是任意表达式。

(3) if 语句中的语句块必须向右缩进，语句块可以是单个语句，也可以是多个语句。当包含两个或两个以上的语句时，语句必须缩进一致，即语句块中的语句必须上下对齐。例如：

```
if x>y:
  t=x
  x=y
  y=t
```

(4) 如果语句块中只有一条语句，if 语句也可以写在同一行上。例如：




```
x=10
if x>0: print(2*x-1)
```

【例 4.1】 输入 3 个整数 x、y、z，请将这 3 个数由小到大输出。

分析：输入 x、y、z，如果 $x>y$，则交换 x 和 y，否则不交换；如果 $x>z$，则交换 x 和 z，否则不交换；如果 $y>z$，则交换 y 和 z，否则不交换。最后输出 x、y、z。

程序如下：

```
x,y,z=eval(input('请输入 x、y、z: '))
if x>y:
   x,y=y,x
if x>z:
   x,z=z,x
if y>z:
   y,z=z,y
print (x,y,z)
```

程序运行结果：

```
请输入 x、y、z:34,156,23
23 34 156
```

4.2 双分支选择结构

可以用 if 语句实现双分支选择结构，其一般格式为：

```
if表达式:
      语句块 1
else:
      语句块 2
```

其语句功能是：先计算表达式的值，若为 True，则执行语句块 1，否则执行语句块 2，语句块 1 或者语句块 2 执行后再执行 if 语句后面的语句。其执行过程如图 4.2 所示。

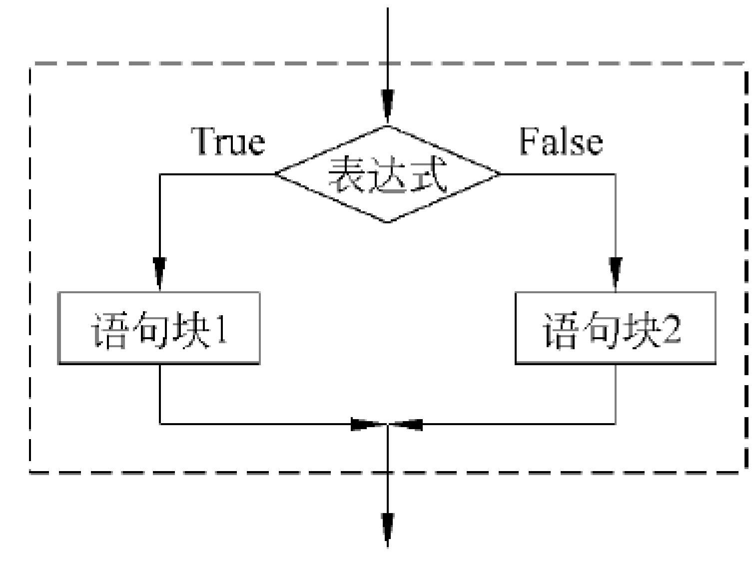
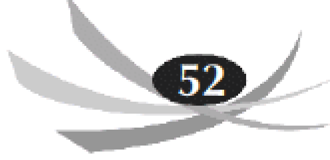


图 4.2 双分支 if 语句的执行过程

注意：与单分支 if 语句一样，对于表达式后面或者 else 后面的语句块，应将它们缩进对齐。例如：

```
if x%2==0:
    y=x+y
    x=x+1
else:
    y=2*x
    x=x-1
```



【例 4.2】 输入年份，判断是否是闰年。

分析：题目的关键是判断闰年的条件，如果年份能被 4 整除但不能被 100 整除或者能被 400 整除，则是闰年，否则就不是闰年。

程序如下：

```
year=int(input('请输入年份：'))
if (year%4==0 and year%100!=0) or (year%400==0):
    print(year,'年是闰年')
else:
    print(year,'年不是闰年')
```

程序运行结果：

```
请输入年份：2017
2017年不是闰年
```

再次运行程序，结果如下：

```
请输入年份：2000
2000年是闰年
```

4.3 多分支选择结构

多分支 if 语句的一般格式为：

```
if表达式 1:
   语句块 1
elif表达式 2:
   语句块 2
elif表达式 3:
   语句块 3
   …
elif表达式 m:
   语句块 m
[else:
   语句块 n]
```

其语句功能是：当表达式 1 的值为 True 时，执行语句块 1，否则求表达式 2 的值；当表达式 2 的值为 True 时，执行语句块 2，否则求表达式 3 的值；以此类推。若表达式的值都为 False，则执行 else 后的语句 n。不管有几个分支，程序执行完一个分支后，其余分支将不再执行。多分支 if 语句的执行过程如图 4.3 所示。

【例 4.3】 输入学生的成绩，根据成绩进行分类，85 分以上为优秀，70～84 分为良好，60～69 分为及格，60 分以下为不及格。

分析：将学生成绩分为四个分数段，然后根据各分数段的成绩，输出不同的等级。程

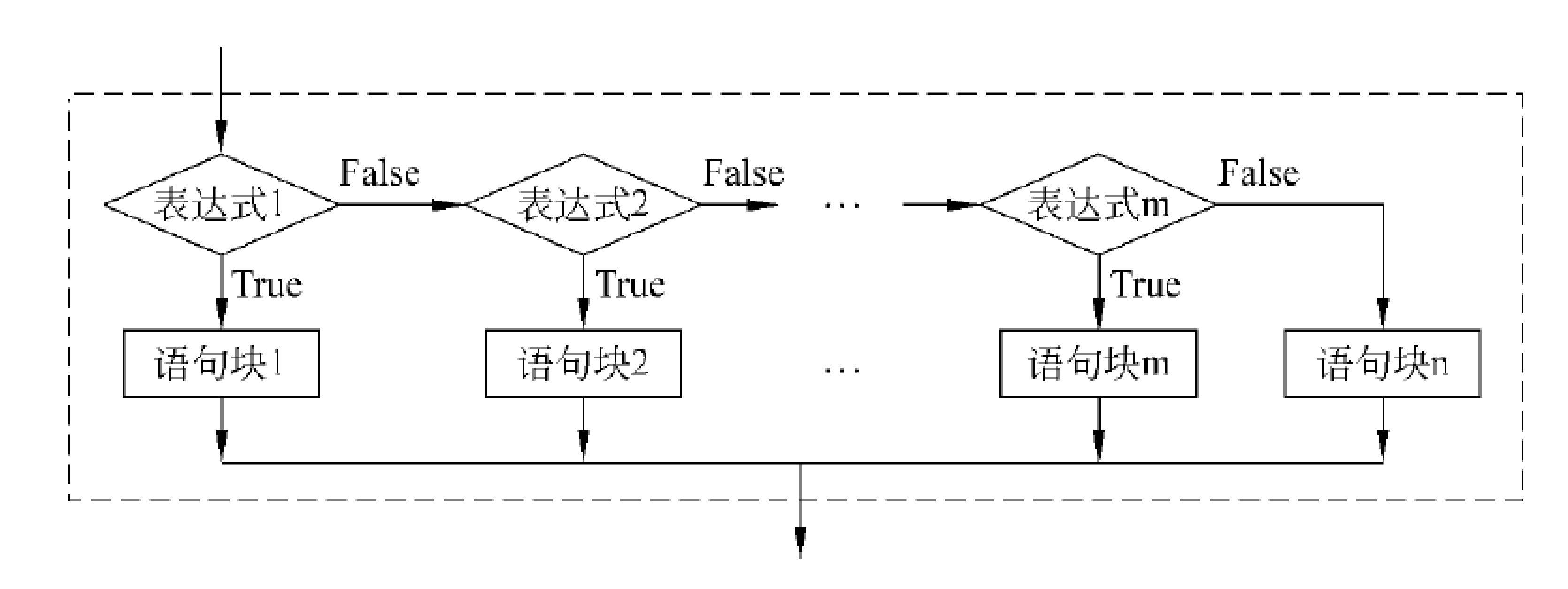


图 4.3　多分支 if 语句的执行过程

序分为四个分支,可以用四个单分支结构实现,也可以用多分支 if 语句实现。

程序如下:

```
score= input("请输入学生成绩:")
ifscore < 60:
    print("不及格")
elifscore < 70:
    print("及格")
elifscore < 85:
    print("良好")
else:
    print("优秀")
```

程序运行结果:

```
请输入学生成绩:83
良好
```

【例 4.4】　从键盘输入一个字符 ch,判断它是英文字母、数字或其他字符。

分析:本题应进行三种情况的判断。

(1) 英文字母:ch>="a" and ch<="z" or ch>="A" and ch<="Z"

(2) 数字字符:ch>="0" and ch<="9"

(3) 其他字符。

程序如下:

```
ch= input("请输入一个字符:")
if ch>="a" and ch<="z" or ch>="A" and ch<="Z":
    print("%c是英文字母"% ch)
elif ch>="0" and ch<="9":
    print("%c是数字"% ch)
else:
    print("%c是其他字符"% ch)
```

程序运行结果:

```
请输入一个字符:L
L是英文字母
```

再次运行程序,结果如下:

```
请输入一个字符:#
#是其他字符
```

4.4 选择结构嵌套

if 语句中可以再嵌套 if 语句,可以有以下不同形式的嵌套结构。

语句一:

```
if 表达式 1:
    if 表达式 2:
      语句块 1
    else:
语句块 2
```

语句二:

```
if 表达式 1:
    if 表达式 2:
      语句块 1
else:
    语句块 2
```

Python 根据对齐关系来确定 if 之间的逻辑关系,在语句一中,else 与第二个 if 匹配,在语句二中 else 与第一个 if 匹配。

【例 4.5】 选择结构的嵌套问题。

购买地铁车票的规定如下:乘 1~4 站,3 元/位;乘 5~9 站,4 元/位;乘 9 站以上,5 元/位。输入人数、站数,输出应付款。

分析:需要进行两次分支。根据"人数<=4"分支一次,表达式为假时,还需要根据"人数<=9"分支一次。流程图如图 4.4 所示。

程序如下:

```
n,m=input('请输入人数,站数:')
if m<=4:
    pay=3*n
else:
    if m<=9:
        pay=4*n
    else:
        pay=5*n
```

```
print('应付款：',pay)
```

程序运行结果：

```
请输入人数,站数：3,5
应付款：12
```

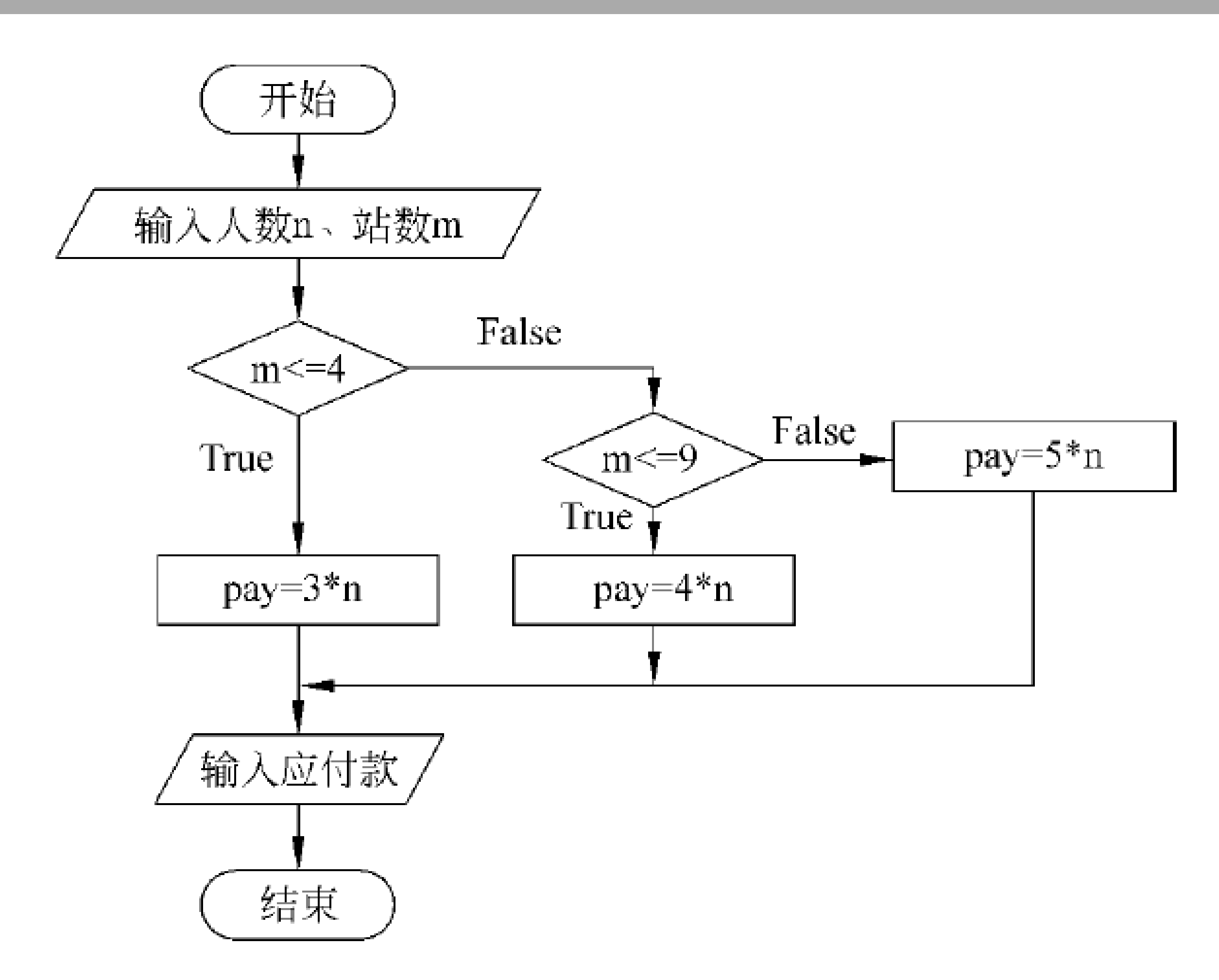


图 4.4　计算乘地铁应付款流程图

【例 4.6】　求一元二次方程 $ax^2+bx+c=0$ 的根。

程序如下：

```
import math
a,b,c=eval(input("请输入一元二次方程的系数："))
if a==0:
    print('输入错误!')
else:
    delta=b*b-4*a*c
    x=-b/(2*a)
    if delta==0:
        print('方程有唯一解,X=%f'%(x))
    elif delta>0:
        x1=x-math.sqrt(delta)/(2*a)
        x2=x+math.sqrt(delta)/(2*a)
        print('方程有两个实根:X1=%f,X2=%f'%(x1,x2))
    else:
        x1=(-b+complex(0,1)*math.sqrt((-1)*delta))/(2*a)
        x2=(-b-complex(0,1)*math.sqrt((-1)*delta))/(2*a)
        print('方程有两个虚根,分别是：')
        print(x1,x2)
```

程序运行结果：

```
请输入一元二次方程的系数:0,1,1
输入错误!
```

再次运行程序,结果如下:

```
请输入一元二次方程的系数:1,2,1
方程有惟一解,x=-1.000000
```

再次运行程序,结果如下:

```
请输入一元二次方程的系数:5,2,3
方程有两个虚根,分别是:
(-0.2+0.7483314773547882j) (-0.2-0.7483314773547882j)
```

4.5 选择结构程序举例

选择结构在执行时依据一定的条件选择程序的执行路径,程序设计的关键在于构造合适的分支条件和分析程序流程,根据不同的程序流程选择适当的分支语句。为了加深对选择结构程序设计方法的理解,下面再看几个例子。

【例 4.7】 从键盘输入一个实数,不调用 math.h 中的库函数计算其绝对值和平方值并输出。

程序如下:

```
a=float(input('input:'))
if a>=0:
    b=a
else:
    b=-a
c=a**2
print("abs=%f,square=%f"%(b,c))
```

程序运行结果:

```
input:-5
abs=5.000000,square=25.000000
```

【例 4.8】 输入三角形的三条边长,求三角形的面积。

分析:设 a,b,c 表示三角形的三条边长,则构成三角形的充分必要条件是任意两边之和大于第三边,即 $a+b>c,b+c>a,c+a>b$。如果该条件满足,则可按照海伦公式计算三角形的面积:

$$s=\sqrt{p(p-a)(p-b)(p-e)}$$

其中,$p=(a+b+c)/2$。

程序如下:

```
from math import *
a,b,c=eval(input('a,b,c='))
if a+b>c and a+c>b and b+c>a:
    p=(a+b+c)/2
    s=sqrt(p*(p-a)*(p-b)*(p-c))
    print('area=',s)
else:
    print('input data error')
```

程序运行结果：

```
a,b,c=3,4,5
area=6.0
```

【例 4.9】 输入一个整数，判断它是否为水仙花数。所谓水仙花数，是指这样的一些三位整数：各位数字的立方和等于该数本身，例如 $153=1^3+5^3+3^3$，因此 153 是水仙花数。

分析：题目的关键是先分别求出这个三位整数的个位、十位和百位数字，再根据判定条件判断该数是否为水仙花数。

程序如下：

```
x=input('请输入三位整数 x: ')
a=x//100
b=(x-a*100)//10
c=x-100*a-10*b
if x==a**3+b**3+c**3:
    print(x,'是水仙花数')
else:
print(x,'不是水仙花数')
```

程序运行结果：

```
请输入三位整数 x: 153
153是水仙花数
```

【例 4.10】 某运输公司的收费按照用户运送货物的路程进行计算，其运费折扣标准如表 4.1 所示。请编写程序计算运输公司的计费。

表 4.1 运输公司运费计算方法

路程/km	运费的折扣	路程/km	运费的折扣
$s<250$	没有折扣	$1000\leqslant s<2000$	8%
$250\leqslant s<500$	2%	$2000\leqslant s<3000$	10%
$500\leqslant s<1000$	5%	$s\geqslant 3000$	15%

分析：首先要输入路程，然后根据路程决定运费折扣是多少，再进行运费的计算。

程序如下：

```
s= input("请输入路程：")
if
    s< 250:
    fee= s
if 250<=s< 500:
    fee= (0.98) * s
if 500<=s< 1000:
    fee= (0.95) * s
if 1000<=s< 2000:
    fee= (0.92) * s
if 2000<=s< 3000:
    fee= (0.90) * s
if 3000<=s:
    fee= (0.85) * s
print('运费是：%.6f'% sum)
```

程序运行结果：

```
请输入路程：300
运费是：294.0
```

【例 4.11】 如图 4.5 所示，在直角坐标系中有一个以原点为中心的单位圆，今任给一点(x,y)，试判断该点是在单位圆内、单位圆上，还是单位圆外。若在单位圆外，那么是在 x 轴的上方、下方，还是在 x 轴上？

图 4.5　单位圆与点示意图

分析：以原点为中心的单位圆的方程为 $x^2+y^2=1$，因此，对任意点(x,y)，若 $x^2+y^2<1$，则该点在单位圆内；对任意点(x,y)，若 $x^2+y^2>1$，则该点在单位圆外；对任意点(x,y)，若 $x^2+y^2=1$ 则该点在单位圆上。这就形成了三个分支，而 if 语句只能解决二路分支问题。为此，先将问题变成两个分支，即若 $x^2+y^2\leqslant 1$，则该点在单位圆内或单位圆上；否则该点在单位圆外。当该点在单位圆外时，还要考虑是在 x 轴的下方、上方还是 x 轴上。对这一三分支问题，可以仿照上面的方法将它变成二路分支来处理。根据以上分析，流程图如图 4.6 所示。

程序如下：

```
x,y= eval(input("请输入 x和 y: "))
if x* * 2+ y* * 2<=1:
    if x* * 2+ y* * 2==1:
      print("点 (%f,%f)在单位圆上"% (x,y))
```

```
        else:
            print("点 (%f,%f)在单位圆内"% (x,y))
else:
    if y>=0:
      if y==0:
            print("点 (%f,%f)在单位圆外,在 x轴上"% (x,y))
      else:
            print("点 (%f,%f)在单位圆外,在 x轴上方"% (x,y))
    else:
            print("点 (%f,%f)在单位圆外,在 x轴下方"% (x,y))
```

程序运行结果：

```
请输入 x和 y: 1,0
点 (1.000000,0.000000)在单位圆上
```

再次运行程序,结果如下：

```
请输入 x和 y: 3,-2
点 (3.000000,-2.000000)在单位圆外,在 x轴下方
```

再次运行程序,结果如下：

```
请输入 x和 y: 0.2,0.3
点 (0.200000,0.300000)在单位圆内
```

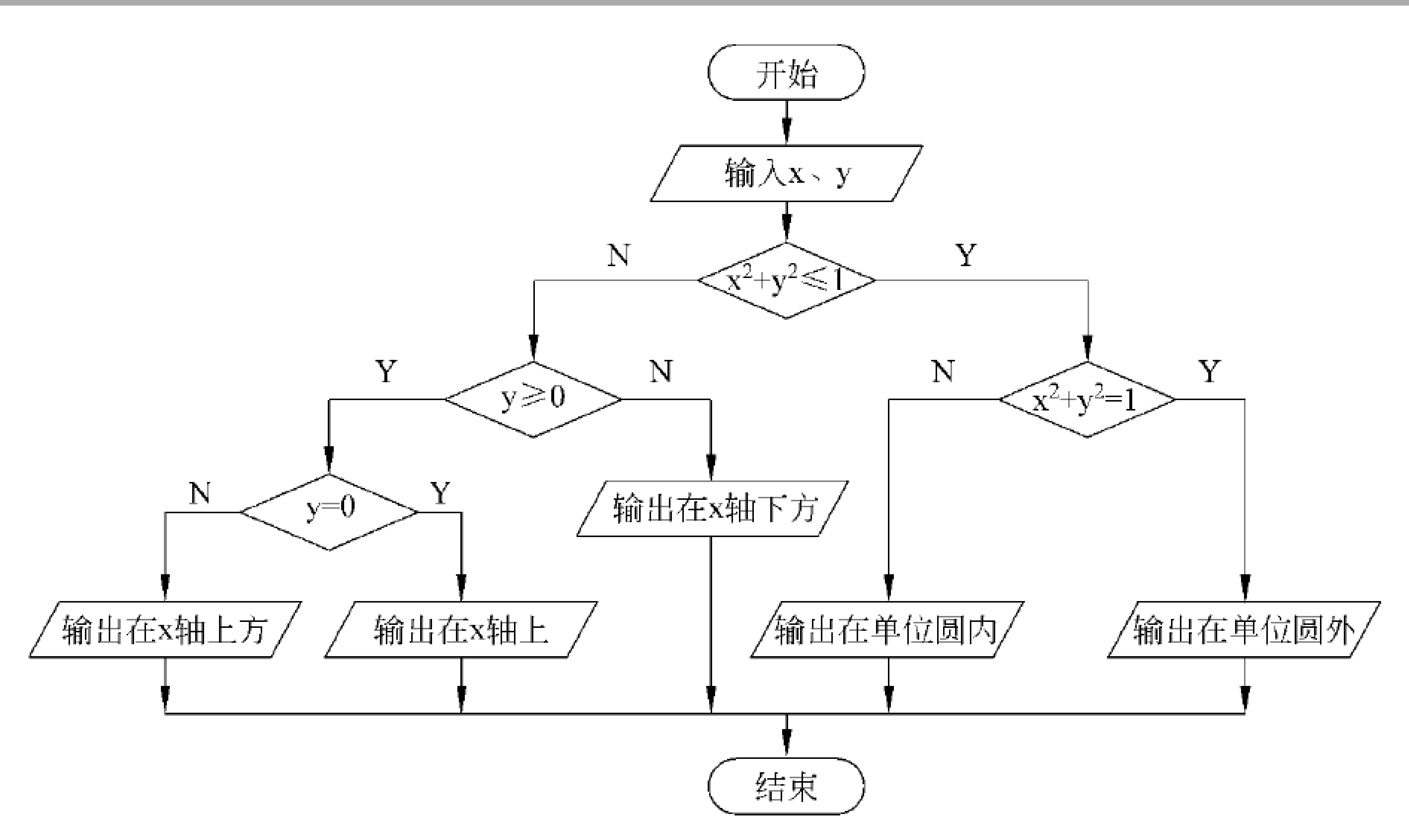


图 4.6　流程图

习　题

1. 选择题

(1) 以下统计“成绩(mark)优秀的男生以及不及格的男生”的人数，正确的语句为(　　)。

A. if (gender=="男" and mark<60 or mark>=90): n+=1

B. if (gender=="男" and mark<60 and mark>=90): n+=1

C. if (gender=="男") and (mark<60 and mark>=90): n+=1

D. if (gender=="男") or mark<60 or mark>=90): n+=1

(2) 用 if 语句表示如下分段函数：

$$y=\begin{cases}x^2-2x+3 & x<1\\ \sqrt{x-1} & x\geqslant 1\end{cases}$$

下面不正确的程序段是(　　)。

A. if (x<1): y=x*x-2*x+3
   else: y=math.sqrt(x-1)

B. if (x<1): y=x*x-2*x+3
   y=math.sqrt(x-1)

C. y=x*x-2*x+3
   if (x>=1): y=math.sqrt(x-1)

D. if (x<1): y=x*x-2*x+3
   if (x>=1): y=math.sqrt(x-1)

(3) 执行下列 Python 语句将产生的结果是(　　)。

```
x=2
y=2.0
if (x==y): print("Equal")
else: print("No Equal")
```

A. Equal　　B. Not Equal

C. 编译错误　　D. 运行时错误

(4) 下列 Python 程序的运行结果是(　　)。

```
x=0
y=True
print(x>y and'A'<'B')
```

A. True　　B. False　　C. 0　　D. 1

2. 填空题

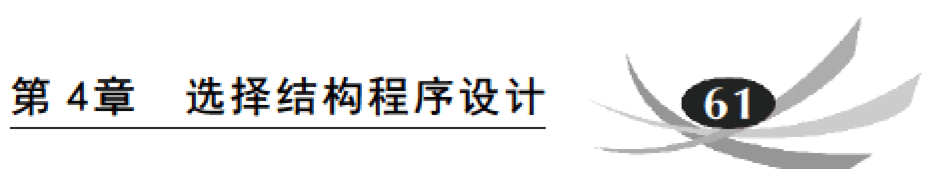
(1) 对于 if 语句中的语句块，应将它们________。

(2) 当 x=0，y=50 时，语句 z=x if x else y 执行后，z 的值是________。

(3) 判断整数 x 奇偶性的 if 条件语句是________。

(4) 说明以下 3 个 if 语句的区别：________。

```
a: if i>0:
```




```
    if j>0 :n=1
    else:n=2
b: if i>0:
    if j>0:n=1
    else:n=2
c: if i>0:n=1
    else:
    if j>0:n=2
```

(5) 下列程序段的功能是________。

```
a=3
b=5
if a>b:
    t=a
    a=b
    b=t
print a,b
```

3. 编程计算函数的值。

$$y=\begin{cases}x+9, & x<-4\\ x^2+2x+1, & -4\leqslant x<4\\ 2x-15, & x\geqslant 4\end{cases}$$

4. 在购买某物品时，若标明的价钱 x 在下面范围内，所付钱 y 按对应折扣支付，其数学表达式如下：

$$y=\begin{cases}x, & x<1000\\ 0.9x, & 1000\leqslant x<2000\\ 0.8x, & 2000\leqslant x<3000\\ 0.7x, & x>3000\end{cases}$$

5. 计算器程序。用户输入运算数和四则运算符，输出计算结果。

6. 数 x、y 和 z，如果 $x^2+y^2+z^2>1000$，则输出 $x^2+y^2+z^2$ 千位以上的数字，否则输出三个数之和。

7. 某公司员工的工资计算方法如下。

(1) 工作时数超过 120 小时者，超过部分加发 15%。

(2) 工作时数低于 60 小时者，扣发 700 元。

(3) 其余按每小时 80 元计发。

输入员工的工号和该员工的工作时数，计算应发工资。

第5章 循环结构程序设计

结构化程序由顺序结构、选择结构和循环结构组成。前面已经介绍了顺序结构和选择结构的程序设计，这一章主要介绍循环结构程序设计。

循环结构是一种重复执行的程序结构。在许多实际问题中，需要对问题的一部分通过若干次有规律的重复计算来实现。例如，求大量的数据之和、迭代求根、递推法求解等，这些都要用到循环结构的程序设计。循环是计算机解题的一个重要特征，计算机运算速度快，最善于进行重复性的工作。

在 Python 中，能用于循环结构的流程控制语句有 while 语句和 for 语句。下面将对这两种循环分别进行介绍。

5.1 while 循环结构

5.1.1 while语句

1. while 语句的一般格式

while 语句的一般格式为：

```
while条件表达式:
    循环体
```

功能：条件表达式描述循环的条件，循环体语句描述要反复执行的操作，称为循环体。while 语句执行时，先计算条件表达式的值，当条件表达式的值为真(非 0)时，循环条件成立，执行循环体；当条件表达式的值为假(0)时，循环条件不成立，退出循环，执行循环语句的下一条语句。其执行流程如图 5.1 所示。

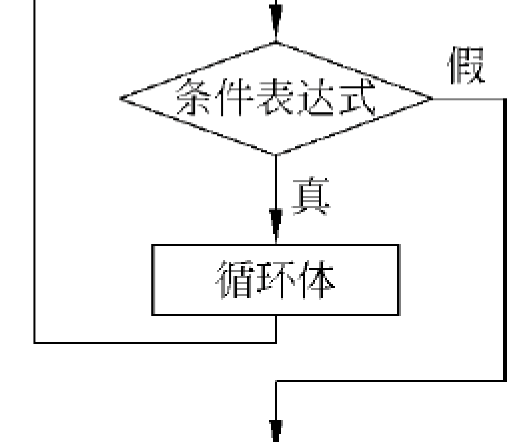


图 5.1 while 循环流程图

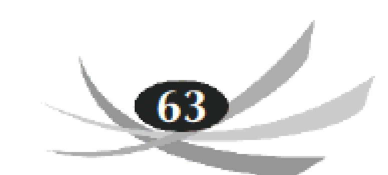
注意：

(1) 当循环体由多条语句构成时，必须用缩进对齐的方式组成一个语句块来分隔子句，否则会产生错误。

(2) 与 if 语句的语法类似，如果 while 循环体中只有一条语句，可以将该语句与 while 写在同一行中。

(3) while 语句的条件表达式不需要用括号括起来，表达式后面必须有冒号。

(4) 如果表达式永远为真，循环将会无限地执行下去。在循环体内必须有修改表达式值的语句，使其值趋向 False，让循环趋于结束，避免无限循环。



2. 在 while 语句中使用 else 子句

while 语句中使用 else 子句的一般形式为：

```
while 条件表达式:
    循环体
else:
    语句
```

Python 与其他大多数语言不同，可以在循环语句中使用 else 子句，即构成了 while…else 循环结构，else 中的语句会在循环正常执行完的情况下执行（不管是否执行循环体）。例如：

```
count=int(input())
while count<5:
    print(count,"is less han 5")
    count=count+1
else:
    print(count,"is not less than 5")
```

程序的一次运行结果如下：

```
8↙
8 is not less than 5
```

在该程序中，当输入 8 时，循环体一次都没有执行，退出循环时，执行 else 子句。

5.1.2 while 语句应用

【例 5.1】 求 $\sum_{n=1}^{100} n$。

分析：该题目实际是求若干个数的累加问题。定义 sum 存放累加和，用 n 表示加数，用循环结构解决，每循环一次累加一个整数值，整数的取值范围为 1～100。

程序如下：

```
sum,n=0,1
while n<=100:
    sum=sum+n
    n=n+1
print("1+2+3+…+100=",sum)
```

程序运行结果：

```
1+2+3+…+100=5050
```

本程序中变量 n 在本题中有两个作用：一是作为循环计数变量；二是每次被累加的整数值。循环体有两条语句：sum＝sum ＋ n 实现累加；n＝n＋1 使加数 n 每次加 1，这

是改变循环条件的语句，否则循环不能终止，成为“死循环”。循环条件是当 n 小于或等于 100 时，执行循环体，否则跳出循环，执行循环语句的下一条语句（print 语句）以输出计算结果。

思考：如果将循环体语句 s=s+n 和 n=n+1 互换位置，程序应如何修改？

对于 while 语句的用法，还需要注意以下几点。

（1）如果 while 后面表达式的值一开始就为假，则循环体一次也不执行。例如：

```
a=0
b=0
while a>0:
b=b+1
```

（2）循环体中的语句可以是任意类型的语句。

（3）遇到下列情况，退出 while 循环：

① 表达式不成立；

② 循环体内遇到 break、return 语句。

【例 5.2】 从键盘上输入若干个数，求所有正数之和。当输入 0 或负数时，程序结束。

程序如下：

```
sum=0
x=input("请输入一个正整数(输入 0或者负数时结束):")
while x>=0:
    sum=sum+x
    x=input("请输入一个正整数(输入 0或者负数时结束):")
print("sum=",sum)
```

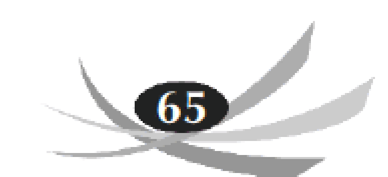
程序运行结果：

```
请输入一个正整数(输入 0或者负数时结束):13
请输入一个正整数(输入 0或者负数时结束):21
请输入一个正整数(输入 0或者负数时结束):5
请输入一个正整数(输入 0或者负数时结束):54
请输入一个正整数(输入 0或者负数时结束):0
sum=92
```

【例 5.3】 输入一个正整数 x，如果 x 满足 $0<x<99\,999$，则输出 x 是几位数并输出 x 个位上的数字。

程序如下：

```
x=int(input("Please input x: "))
if x>=0 and x<99999:
   i=x
   n=0
   while i>0:
```




```
            i=i//10
            n=n+1
    a=x%10
    print("%d是%d位数,它的个位上数字是%d"%(x,n,a))
else:
        print("输入错误!")
```

程序运行结果:

```
Please input x: 12345
12345是 5位数,它的个位上数字是 5
```

再次运行程序,结果如下:

```
Please input x:-1
输入错误!
```

5.2　for 语句结构

5.2.1　for 语句

1. for 语句的一般格式

for 语句是循环控制结构中使用较广泛的一种循环控制语句,特别适合于循环次数确定的情况。其一般格式为:

```
for 目标变量 in 序列对象:
    循环体
```

for 语句的首行定义了目标变量和遍历的序列对象,后面是需要重复执行的语句块。语句块中的语句要向右缩进,且缩进量要一致。

注意:

(1) for 语句是通过遍历任意序列的元素来建立循环的,针对序列的每一个元素执行一次循环体。列表、字符串、元组都是序列,可以利用它们来建立循环。

(2) for 语句也支持一个可选的 else 块,它的功能就像在 while 循环中一样,如果循环离开时没有碰到 break 语句,就会执行 else 块。也就是序列所有元素都被访问过了之后,执行 else 块。其一般格式为:

```
for 目标变量 in 序列对象:
    语句块
else:
        语句
```

2. rang 对象在 for 循环中的应用

在 Python 3.x 中,range()函数返回的是可迭代对象。Python 专门为 for 语句设计



了迭代器的处理方法。range()内建函数的一般格式为：

```
range([start,]stop[,step])
```

range()函数共有三个参数：start 和 step 是可选的，start 表示开始，默认值为 0；end 表示结束；step 表示每次跳跃的间距，默认值为 1。函数功能是生成一个从 start 参数的值开始，到 end 参数的值结束(但不包括 end)的数字序列。

例如，传递一个参数的 range()函数：

```
>>> for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

传递两个参数的 range()函数：

```
>>> for i in range(2,4):
    print(i)
```

```
2
3
```

传递三个参数的 range()函数：

```
>>> for i in range(2,20,3):
    print(i)
```

```
2
5
8
11
14
17
```

执行过程中首先对关键字 in 后的对象调用 iter()函数获得迭代器，然后调用 next()函数获得迭代器的元素，直到抛出 StopIteration 异常。

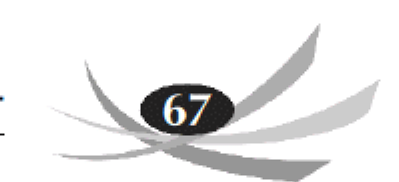
range()函数的工作方式类似于分片。它包含下限，但不包含上限。如果希望下限为 0，则只可以提供上限。例如：

```
>>> range(10)
```

```
[0,1,2,3,4,5,6,7,8,9]
```

【例 5.4】 用 for 循环语句实现例 5.1。

程序如下：




```
sum=0
for i in range(101):
    sum=sum+i
print("1+2+3+…+100=",sum)
```

该例中采用 range()函数得到一个 0～100 的序列，变量 i 依次从序列中取值累加到 sum 变量中。

5.2.2　for 语句应用

【例 5.5】　判断 m 是否为素数。

一个自然数，若除了 1 和它本身外不能被其他整数整除，则称为素数。例如，2，3，5，7，…。根据定义，只需检测 m 是否是被 2，3，4，…，$m-1$ 整除，只要能被其中一个数整除，则 m 不是素数，否则就是素数。

程序中设置标志量 flag，若 flag 为 0 时，则 m 不是素数；若 flag 为 1 时，则 m 是素数。

程序如下：

```
m=int(input("请输入要判断的正整数 m: "))
flag=1
for i in range(2,m):
   if  m%i==0:
      flag=0
if  flag==1:
    print("%d是素数"%m)
else:
    print("%d不是素数"%m)
```

程序运行结果：

```
请输入要判断的正整数 m: 11
11是素数
```

再次运行程序，结果如下：

```
请输入要判断的正整数 m: 20
20不是素数
```

【例 5.6】　已知四位数 3025 具有特殊性质：它的前两位数字 30 与后两位数字 25 之和是 55，而 55 的平方正好等于其本身 3025。编写程序，列举出具有这种性质的所有四位数。

分析：采用列举的方法。将给定的四位数按前两位数、后两位数分别进行分离，验证分离后的两个两位数之和的平方是否等于分离前的那个四位数，若等于即打印输出。

程序如下：

```
print("满足条件的四位数分别是：")
for i in range(1000,10000):
        a=i//100
        b=i%100
        if  (a+b)**2==i:
                print(i)
```

程序运行结果：

```
满足条件的四位数分别是：
2025
3025
9801
```

【例 5.7】 求出 1～100 能被 7 或 11 整除但不能同时被 7 和 11 整除的所有整数并将它们输出。每行输出 10 个。

分析：列举出 1～100 的所有整数，根据题目中的条件对这些数据进行筛选。要控制每行输出 10 个，则应使用 count 变量，用于计数，每当有一个满足条件的数输出时，count 加 1，当 count 能整除 10 时，则换行。

程序如下：

```
print("满足条件的数分别是：")
count=0
for i in range(1,100):
    if i%7==0 and i%11!=0 or i%11==0 and i%7!=0:
        print(i,end="   ")
        count=count+1
        if count%10==0:
            print("")
```

程序运行结果：

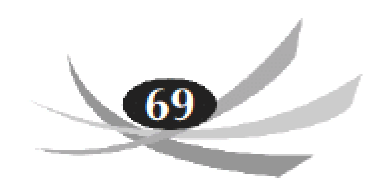
```
满足条件的数分别是：
7   11  14  21  22  28  33  35  42  44
49  55  56  63  66  70  84  88  91  98
99
```

5.3 循环的嵌套

如果一个循环结构的循环体内又包括一个循环结构，就称为循环的嵌套。这种嵌套过程可以有很多重。一个循环外面仅包含一层循环称为两重循环；一个循环外面包含两层循环称为三重循环；一个循环外面包含多层循环称为多重循环。

循环语句 while 和 for 可以相互嵌套。在使用循环嵌套时，应注意以下几个问题。

（1）外层循环和内层循环控制变量不能同名，以免造成混乱。



(2) 循环嵌套的缩进在逻辑上一定要注意,以保证逻辑上的重要性。

(3) 循环嵌套不能交叉,即在一个循环体内必须完整地包含另一个循环。如图 5.2 所示的循环嵌套都是合法的循环嵌套形式。

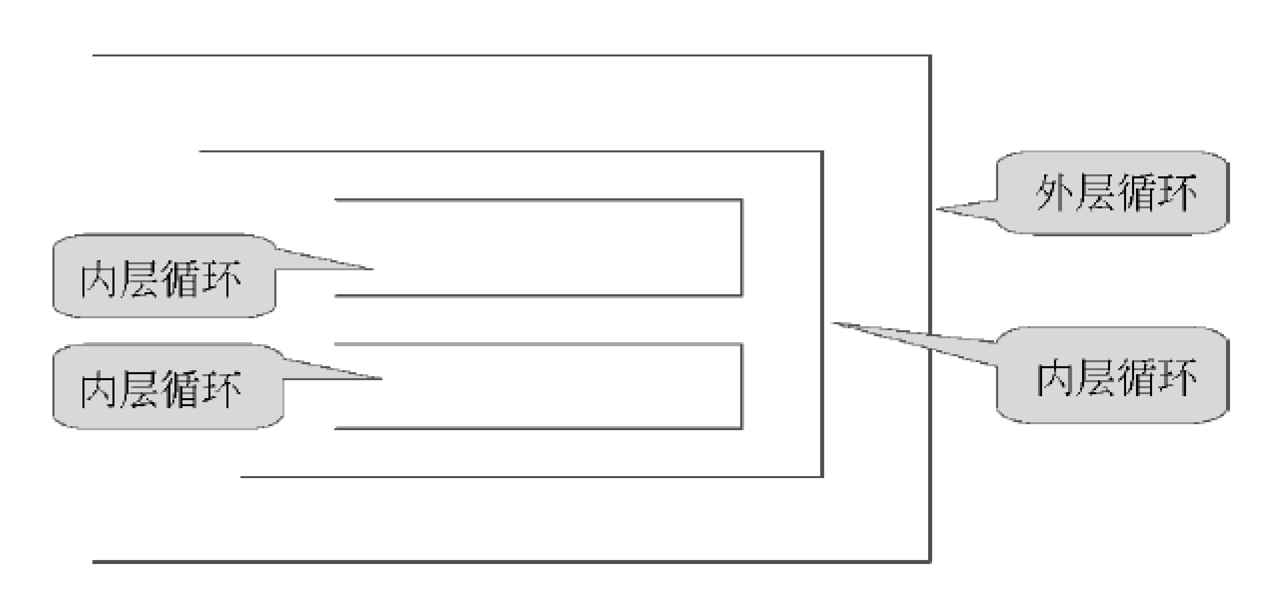


图 5.2　合法的循环嵌套形式

嵌套循环执行时,先由外层循环进入内层循环,并在内层循环终止后接着执行外层循环,再由外层循环进入内层循环中,当内层循环终止时,程序结束。

【例 5.8】 输出九九乘法表,格式如下。

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6   3*3=9
1*4=4  2*4=8   3*4=12  4*4=16
1*5=5  2*5=10  3*5=15  4*5=20  5*5=25
1*6=6  2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7  2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8  2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9  2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

程序如下:

```
for i in range(1,10,1):                  #控制行
    for j in range(1,i+1,1):             #控制列
        print("%d*%d=%-2d  "%(j,i,i*j),end="")
    print("")                            #每行末尾的换行
```

【例 5.9】 找出所有的三位数,要求它的各位数字的立方和正好等于这个三位数。例如 $153=1^3+5^3+3^3$ 就是这样的数。

分析:假设所求的三位数百位数字是 i,十位数字是 j,个位数字是 k,根据题目描述,应满足 $i^3+j^3+k^3=i\times100+j\times10+k$。

程序如下:

```
for i in range(1,10):
    for j in range(0,10):
        for k in range(0,10):
            if i**3+j**3+k**3==i*100+j*10+k:
```

```
print("%d%d%d"%(i,j,k))
```

程序运行结果：

```
153
370
371
407
```

从程序中可以看出，三个 for 语句循环嵌套在一起，第二个 for 语句是前一个 for 语句的循环体，第三个 for 语句是第二个 for 语句的循环体，第三个 for 语句的循环体是 if 语句。

【例 5.10】 求 100～200 的全部素数。

在例 5.5 中可判断给定的整数 m 是否是素数。本例要求 100～200 的所有素数，可在外层加一层循环，用于提供要考查的整数 $m=100,101,\cdots,200$。

程序如下：

```
print("100~200的素数有：")
for m in range(100,201):
        flag=1
        for i in range(2,m):
          if  m%i==0:
              flag=0
              break
        if flag==1:
                print(m,end=" ")
```

程序运行结果：

```
100~200的素数有：
101  103  107  109  113  127  131  137  139  149  151  157  163  167  173  179  181  191  193
197  199
```

5.4 循环控制语句

有时候需要在循环体中提前跳出循环，或者在某种条件满足时，不执行循环体中的某些语句而立即从头开始新一轮的循环，这时就要用到循环控制语句 break、continue 和 pass 语句。

5.4.1 break 语句

break 语句用在循环体内，迫使所在循环立即终止，即跳出所在循环体，继续执行循环结构后面的语句。break 语句对循环执行过程的影响如图 5.3 所示。

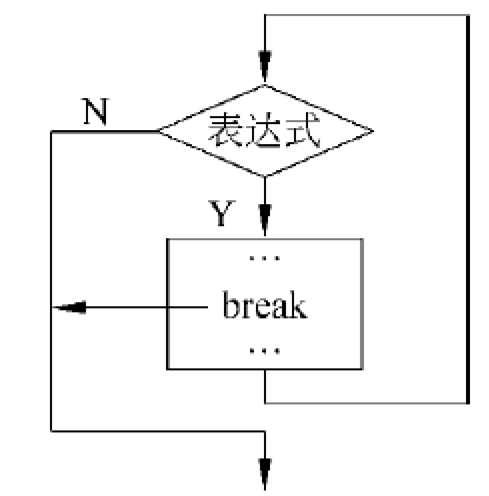
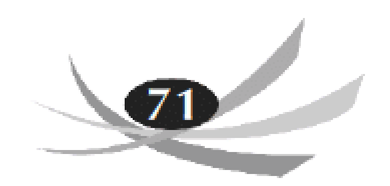


图 5.3 break 语句对循环执行过程的影响示意图

【例 5.11】 求两个整数 a 与 b 的最大公约数。



分析：找出 a 与 b 中较小的一个，则最大公约数必在 1 与较小整数的范围内。使用 for 语句，循环变量 i 从较小整数变化到 1。一旦循环控制变量 i 同时能被 a 与 b 整除，则 i 就是最大公约数，然后使用 break 语句强制退出循环。

程序如下：

```
m,n=eval(input("请输入两个整数："))
if m<n:
      min=m
else:
      min=n
for i in range(min,1,-1):
    if m%i==0 and n%i==0:
         print("最大公约数是：",i)
         break
```

程序运行结果：

```
请输入两个整数：156,18
最大公约数是：6
```

注意：

(1) break 语句只能用于由 while 或 for 语句构成的循环结构中。

(2) 在循环嵌套的情况下，break 语句只能终止并跳出包含它的最近的一层循环体。

5.4.2　continue 语句

当在循环结构中遇到 continue 语句时，程序将跳过 continue 语句后面尚未执行的语句，重新开始下一轮循环，即只结束本次循环的执行，并不终止整个循环的执行。continue 语句对循环执行过程的影响如图 5.4 所示。

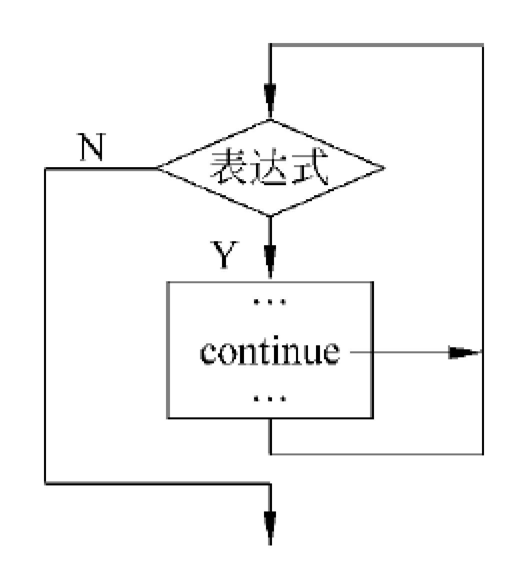


图 5.4　continue 语句对循环执行过程的影响示意图

【例 5.12】　求 1～100 的全部奇数之和。

程序如下：

```
x=y=0
while True:
  x+=1
```

```
    if not (x% 2):continue                # x为偶数直接进行下一次循环
    elif x> 100:break                     # x> 100时退出循环
    else:y+ = x                           # 实现累加
print ("y= ",y)
```

程序运行结果：

```
y= 2500
```

5.4.3 pass语句

pass语句是一个空语句，它不做任何操作，代表一个空操作，在特别的时候用来保证格式或是语义的完整性。例如下面的循环语句：

```
for i in range(5):
    pass
```

该语句的确会循环5次，但是除了循环本身之外，它什么也没做。

【例5.13】 pass语句应用：逐个输出"Python"字符串中的字符。

程序如下：

```
for letter in "Python":
    if letter == "o":
        pass
        print ("This is pass block")
    print ("Current Letter :",letter)
print ("End!")
```

程序运行结果：

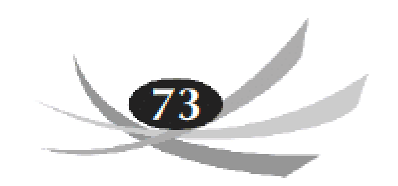
```
Current Letter: P
Current Letter: y
Current Letter: t
Current Letter: h
This is pass block
Current Letter: o
Current Letter: n
End!
```

在程序中，当遇到字母o时，执行pass语句，接着执行print("This is pass block")语句。从运行结果可以看出，pass语句对其他语句的执行没有产生任何影响。

5.5 循环结构程序举例

【例5.14】 利用下面的公式求π的近似值，要求累加到最后一项小于10^{-6}为止。

$$\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$



分析：这是一个累加求和的问题，但是这里的循环次数是预先未知的，而且累加项正负交替出现。如何解决这类问题呢？

在本例中，累加项的构成规律可用寻找累加项通式的方法得到。具体可表示为通式 t=s/n，即累加项由分子和分母两部分组成。分子 s 为+1，-1，+1，-1，…，交替变化，可以采用赋值语句 s=-s 实现，s 的初始值取 1；分母 n 为 1，3，5，7，…的规律递增，可采用 n=n+2 实现，n 的初始值取 1.0。

程序如下：

```
import math
s=1
n=1.0
t=1.0
pi=0
while math.fabs(t)>=1e-6:
        pi=pi+t
        n=n+2
        s=-s
     t=s/n
pi=pi*4
print("PI=%f"%pi)
```

程序运行结果：

```
PI=3.141591
```

【例 5.15】 两个乒乓球队进行比赛，各出三人。甲队为 a，b，c 三人，乙队为 x，y，z 三人。以抽签决定比赛名单。有人向队员打听比赛的名单，a 说他不和 x 比，c 说他不和 x，z 比。编写程序找出三队比赛对手的名单。

分析：可采用枚举的方法实现。

程序如下：

```
for i in range(ord('x'),ord('z')+1):
  for j in range(ord('x'),ord('z')+1):
      if i!=j:
          for k in range(ord('x'),ord('z')+1):
              if (i!=k) and (j !=k):
                  if (i!=ord('x')) and (k!=ord('x')) and (k!=ord('z')):
                      print('order is:\na-->%s\nb-->%s\nc-->%s' % (chr(i),chr(j),chr(k)))
```

程序运行结果：

```
order is:
a-->z
b-->x
c-->y
```

【例 5.16】 正弦函数的泰勒展开式是 $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$，编程计算 $\sin x$ 的值，要求最后一项的绝对值小于 10^{-7}。

分析：根据泰勒展开式，设通项式为 fitem，它由分子(e)、分母(d)和符号(s)三部分组成。可以得出 $\sin x$ 的通项公式有如下特点。

(1) 分子：$e_0 = x, e_i = e_{i-1} \times x \times x$；

(2) 分母：$d_0 = 1, d_i = d_{i-1} \times (n+1) \times (n+2)$，$n$ 的初始值取 1；

(3) 符号：$s_0 = 1, s_i = -s_{i-1}$。

程序如下：

```
s=1
i=1
a=int(input("请输入角度值(单位:度):"))
x=3.1415926/180*a                    #将角度转化为弧度
sinx=x
fitem=e=x                            #第 0项为 x,分子即为 x
d=1                                  #第 0项分母为 1
while(fitem>10**-7):
    e=e*x*x
    d=d*(i+1)*(i+2)
    i=i+2
    fitem=e/d                        #求通项的绝对值
    s=-s                             #求该项的符号
    sinx+=s*fitem                    #求正弦值
print("sin(%3.1f)=%.3f"%(a,sinx))
```

程序运行结果：

```
请输入角度值(单位:度):30
sin(30.0)=0.500
```

【例 5.17】 “百钱百鸡”问题。

公鸡 5 文钱一只，母鸡 3 文钱一只，小鸡 3 只一文钱，用 100 文钱买 100 只鸡，其中公鸡、母鸡、小鸡都必须要有，问公鸡、母鸡、小鸡要买多少只刚好凑足 100 文钱？

分析：显然这是一个组合问题，也可以看作是解不定方程的问题，采用列举的方法实现。令 i、j、k 分别表示公鸡、母鸡和小鸡的数目。

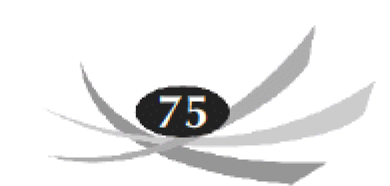
为了确定取值范围，可以有不同的思路，因而也有不同的实现方法，其计算量可能相差甚远。

方法一 令 i、j、k 的列举范围分别如下。

i：1～20(公鸡最多能买 20 只)。

j：1～33(母鸡最多能买 33 只)。

k：1～100(小鸡最多能买 100 只)。



可以采用三重循环逐个搜索。

程序如下：

```
for i in range(1,21):
        for j in range(1,34):
             for k in range(1,101):
                  if i+j+k==100 and i*5+j*3+k/3==100:
                       print("公鸡:%d只,母鸡:%d只,小鸡:%d只"%(i,j,k))
```

程序运行结果：

```
公鸡:4只,母鸡:18只,小鸡:78只
公鸡:8只,母鸡:11只,小鸡:81只
公鸡:12只,母鸡:4只,小鸡:84只
```

在程序中，循环体被执行了 20×33×100 次＝66 000 次。

方法二　令 i、j、k 的列举范围分别如下(保证每种鸡至少买一只)。

i：1～18(公鸡最多能买 18 只)。

j：1～31(母鸡最多能买 31 只)。

k：$100-i-j$(当公鸡和小鸡数量确定后，小鸡的数量可计算得到)。

可以采用两重循环逐个搜索。

程序如下：

```
for i in range(1,19):
          for j in range(1,32):
               k=100-i-j
                    if i+j+k==100 and i*5+j*3+k/3==100:
                         print("公鸡:%d只,母鸡:%d只,小鸡:%d只"%(i,j,k))
```

在程序中，循环体被执行了 18×81＝558 次。

方法三　从题意可得到下列方程组：

$$\begin{cases} i+j+k=100 \\ 5i+3j+\dfrac{k}{3}=100 \end{cases}$$

由方程组可得到式子 $7i+4j=100$。由于 i 和 j 至少为 1，因此可知 i 最大为 13，j 最大为 23。方法二的两重循环可改进为以下程序：

```
for i in range(1,14):
        for j in range(1,24):
                  k=100-i-j
                  if i+j+k==100 and i*5+j*3+k/3==100:
                            print("公鸡:%d只,母鸡:%d只,小鸡:%d只"%(i,j,k))
```

该程序的循环体被执行了 14×24 次＝336 次。

方法四　由方法三中的式 $7i+4j=100$ 可得：$j=(100-7i)/4$。观察 $7i+4j=100$，

$4j$ 同 100 都是 4 的倍数，因此 i 一定也是 4 的倍数。有了这些条件，程序实现时只需要对 i 进行逐个搜索即可，i 的搜索范围为 1～13。

采用单层循环进行逐个搜索。

程序如下：

```
for i in range(1,14):
    j=(100-7*i)/4
    k=100-i-j
    if i%4==0:
        print("公鸡:%d只,母鸡:%d只,小鸡:%d只"%(i,j,k))
```

该算法程序只循环了 13 次。

上述四种方法都能得到相同的运行结果，从程序执行次数分析可知，由于程序的搜索策略不同，程序的计算量也不同。

习　　题

1. 选择题

(1) 以下 for 语句中，(　　)不能完成 1～10 的累加功能。

A. for i in range(10,0):sum+=i

B. for i in range(1,11):sum+=i

C. for i in range(10,0,-1):sum+=i

D. for i in range(10,9,8,7,6,5,4,3,2,1):sum+=i

(2) 设有如下程序段：

```
k=10
while k:
    k=k-1
    print(k)
```

则下面描述中正确的是(　　)。

A. while 循环执行 10 次　　B. 循环是无限循环

C. 循环体语句一次也不执行　　D. 循环体语句执行一次

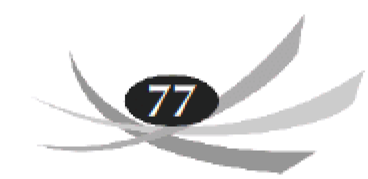
(3) 以下 while 语句中的表达式“not E”等价于(　　)。

```
while not E:
     pass
```

A. E==0　　B. E!=1　　C. E!=0　　D. E==1

(4) 下列程序的结果是(　　)。

```
sum=0
for i in range(100):
    if(i%10):
```




```
        continue
    sum=sum+i
print(sum)
```

A. 5050　　B. 4950　　C. 450　　D. 10

(5) 下列 for 循环执行后，输出结果的最后一行是(　　)。

```
for i in range(1,3):
    for j in range(2,5):
      print(i * j)
```

A. 2　　B. 6　　C. 8　　D. 15

(6) 下列说法中正确的是(　　)。

A. break 用在 for 语句中，而 continue 用在 while 语句中

B. break 用在 while 语句中，而 continue 用在 for 语句中

C. continue 能结束循环，而 break 只能结束本次循环

D. break 能结束循环，而 continue 只能结束本次循环

2. 填空题

(1) Python 提供了两种基本的循环结构：________和________。

(2) 循环语句 for i in range(−3,21,4)的循环次数为________。

(3) 要使语句 for i in range(_,−4,−2)循环执行 15 次，则循环变量 i 的初值应当为________。

(4) 执行下列 Python 语句后的输出结果是________，循环执行了________次。

```
i=-1
while(i<0):
    i*=i
print(i)
```

(5) 当循环结构的循环体由多个语句构成时，必须用________________的方式组成一个语句块。

(6) Python 无穷循环“while True:”的循环体中可用________语句退出循环。

3. 一个五位数，判断它是不是回文数。即个位与万位相同，十位与千位相同，如 12321 是回文数。

4. 求 1+2!+3!+…+20!的和。

5. 求 200 以内能被 11 整除的所有正整数，并统计满足条件的数的个数。

6. 编写一个程序，求 e 的值，当通项小于 10^{-7} 停止计算。

$$e\approx 1+\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{n!}$$

第6章 序　列

序列是程序设计中最基本的数据结构，几乎每一种程序设计语言都提供了类似的数据结构，例如C语言和Visual Basic中的数组等。序列是一系列连续值，这些值通常是相关的，并且按照一定顺序排序。Python提供的序列类型使用灵活，功能强大。Python常用的序列类型结构有列表、元组、字符串、字典、集合等。本章主要介绍前三种，字典和集合将在第7章进行介绍。

序列中的每一个元素都有自己的位置编号，可以通过偏移量索引来读取数据。图6.1是一个包含11个元素的序列。第一个元素，索引为0；第二个元素，索引为1；以此类推。也可以从最后一个元素开始计数，最后一个元素的索引是−1，倒数第二个元素的索引就是−2，以此类推。可以通过索引获取序列中的元素，其一般格式为：

```
序列名[索引]
```

其中，索引又称下标或位置编号，必须是整数或整型表达式。在包含了n个元素的序列中，索引的取值为$0,1,2,\cdots,n-1$和$-1,-2,-3,-\cdots,-n$，即范围为$-n\sim n-1$。

字符	H	e	l	l	o		W	o	r	l	d
索引	0	1	2	3	4	5	6	7	8	9	10
索引	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

图6.1　序列元素与索引对应图

6.1　列　　表

列表(list)是Python中重要的内置数据类型，列表是一个元素的有序集合，一个列表中元素的数据类型可以各不相同，所有元素放在一对方括号“[”和“]”中，相邻元素之间用逗号分隔开。例如：

```
[1,2,3,4,5]
['Python','C','HTML','Java','Perl']
['wade',3.0,81,[ 'bosh','haslem']]        #列表中嵌套了列表类型
```

6.1.1　列表的基本操作

1. 列表的创建

使用赋值运算符“＝”将一个列表赋值给变量即可创建列表对象。例如：

```
>>>a_list=['physics','chemistry',2017,2.5]
>>>b_list=['wade',3.0,81,['bosh','haslem']]        #列表中嵌套了列表类型
>>>c_list=[1,2,(3.0,'hello world!')]               #列表中嵌套了元组类型
>>>d_list=[]                                       #创建一个空列表
```

2. 列表元素读取

使用索引可以直接访问列表元素，方法为：列表名[索引]。如果指定索引不存在，则提示下标越界。例如：

```
>>>a_list=['physics','chemistry',2017,2.5,[0.5,3]]
>>>a_list[1]
'chemistry'
>>>a_list[-1]
[0.5,3]
>>>a_list[5]
Traceback (most recent call last):
  File "<pyshell#9>",line 1,in <module>
    a_list[5]
IndexError: list index out of range
```

3. 列表切片

可以使用“列表序号对”的方法来截取列表中的任意部分，得到一个新列表，称为列表的切片操作。切片操作的方法是：

列表名[开始索引:结束索引:步长]

开始索引：表示是第一个元素对象，正索引位置默认为 0；负索引位置默认为－len(consequence)。

结束索引：表示是最后一个元素对象，正索引位置默认为 len(consequence)－1；负索引位置默认为 －1。

步长：表示取值的步长，默认为 1，步长值不能为 0。

例如：

```
>>>a_list[1:3]                  #开始为 1,结束为 2,不包括 3,步长默认为 1
['chemistry',2017]
>>>a_list[1:-1]
['chemistry',2017,2.5]
>>>a_list[:3]                   #左索引默认为 0
```

```
['physics','chemistry',2017]
>>> a_list[1:]                                #从第一个元素开始截取列表
['chemistry',2017,2.5,[0.5,3]]
>>> a_list[:]                                 #左右索引均缺省
['physics','chemistry',2017,2.5,[0.5,3]]
>>> a_list[::2]                               #左右索引均缺省,步长为 2
['physics',2017,[0.5,3]]
```

4. 增加元素

在实际应用中,列表元素的增加和删除操作也是经常遇到的操作,Python 提供了多种不同的方法来实现这一功能。

(1) 使用"+"运算符将一个新列表添加在原列表的尾部。

例如:

```
>>> id(a_list)                                #获取列表 a_list 的地址
49411096
>>> a_list=a_list+[5]
>>> a_list
['physics','chemistry',2017,2.5,[0.5,3],5]
>>> id(a_list)                                #获取添加元组时 a_list 的地址
49844992
```

从上面的例子可以看出,"+"运算符在形式上实现了列表元素的增加,但从增加前后列表的地址看,这种方法并不是真的为原列表添加元素,而是创建了一个新列表,并将原列表和增加列表依次复制到新创建列表的内存空间。由于需要进行大量元素的复制,因此该方法操作速度较慢,添加大量元素时不建议使用该方法。

(2) 使用列表对象的 append()方法向列表尾部添加一个新的元素。这种方法在原地址上进行操作,速度较快。例如:

```
>>> a_list.append('Python')
>>> a_list
['physics','chemistry',2017,2.5,[0.5,3],5,'Python']
```

(3) 使用列表对象的 extend()方法将一个新列表添加在原列表的尾部。与"+"的方法不同,这种方法在原列表地址上操作。例如:

```
>>> a_list.extend([2017,'C'])
>>> a_list
['physics','chemistry',2017,2.5,[0.5,3],5,'Python',2017,'C']
```

(4) 使用列表对象的 insert()方法将一个元素插入到列表的指定位置。该方法有两个参数:第一个参数为插入位置;第二个参数为插入元素。例如:

```
>>> a_list.insert(1,3.5)                      #插入位置为位置编号
```

```
>>> a_list
['physics',2017,'chemistry',3.5,2.5,[0.5,3],5,'Python',2017,'C']
```

5. 检索元素

(1) 使用列表对象的 index()方法可以获取指定元素首次出现的下标，语法为：index(value,[,start,[,end]])，其中 start 和 end 分别用来指定检索的开始和结束位置，start 默认为 0，end 默认为列表长度。例如：

```
>>> a_list.index(2017)          #在 a_list 列表中检索
1
>>> a_list.index(2017,2)        #从 a_list 列表第 2 个元素开始进行检索
8
>>> a_list.index(2017,5,7)      #在 a_list 列表第 5~7 个元素中检索
Traceback (most recent call last):
  File "<pyshell#10>",line 1,in <module>
    a_list.index(2017,5,7)
ValueError: 2017 is not in list          #在指定范围中没有检索到元素,提示错误信息
```

(2) 使用列表对象的 count()方法统计列表中指定元素出现的次数。例如：

```
>>> a_list.count(2017)
2
>>> a_list.count([0.5,3])
1
>>> a_list.count(0.5)
0
```

(3) 使用 in 运算符检索某个元素是否在该列表中。如果元素在列表中，返回 True，否则返回 False。

```
>>> 5 in a_list
True
>>> 0.5 in a_list
False
```

6. 删除元素

(1) 使用 del 命令删除列表中指定位置的元素。例如：

```
>>> del a_list[2]
>>> a_list
['physics',2017,3.5,2.5,[0.5,3],5,'Python',2017,'C']
```

执行 del a_list[2]后，a_list 中位置编号为 2 的元素被删除，该元素后面的元素自动前移一个位置。

del 命令也可以直接删除整个列表。例如：

```
>>>b_list= [10,7,1.5]
>>>b_list
[10,7,1.5]
>>>del b_list
>>>b_list
Traceback (most recent call last):
  File "<pyshell# 42>",line 1,in <module>
    b_list
NameError: name 'b_list' is not defined
```

删除对象 b_list 之后，该对象就不存在了，再次访问就会提示出错。

(2) 使用列表对象的 remove()方法删除首次出现的指定元素，如果列表中不存在要删除的元素，提示出错信息。例如：

```
>>>a_list.remove(2017)
>>>a_list
['physics',3.5,2.5,[0.5,3],5,'Python',2017,'C']
>>>a_list.remove(2017)
>>>a_list
['physics',3.5,2.5,[0.5,3],5,'Python','C']
>>>a_list.remove(2017)
Traceback (most recent call last):
  File "<pyshell# 30>",line 1,in <module>
    a_list.remove(2017)
ValueError: list.remove(x): x not in list
```

执行第一个 a_list. remove(2017)，删除了第一个 2017，a_list 内容变为['physics', 3.5, 2.5, [0.5, 3], 5, 'Python', 2017, 'C']；执行第二个 a_list. remove(2017)，删除了第二个 2017，a_list 内容变为['physics', 3.5, 2.5, [0.5, 3], 5, 'Python', 'C']；执行第三个 a_list. remove(2017)，系统提示出错。

(3) 使用列表的 pop()方法删除并返回指定位置上的元素，缺省参数时删除最后一个位置上的元素，如果给定的索引超出了列表的范围，则提示出错。例如：

```
>>>a_list.pop()
'C'
>>>a_list
['physics',3.5,2.5,[0.5,3],5,'Python']
>>>a_list.pop(1)
3.5
>>>a_list
['physics',2.5,[0.5,3],5,'Python']
```

```
>>> a_list.pop(5)
Traceback (most recent call last):
  File "<pyshell# 35>",line 1,in <module>
    a_list.pop(5)
IndexError: pop index out of range
```

6.1.2 列表的常用函数

1. cmp()

格式：cmp(列表 1,列表 2)

功能：对两个列表逐项进行比较，先比较列表的第一个元素，若相同则分别取两个列表的下一个元素进行比较；若不同则终止比较。如果第一个列表最后比较的元素大于第二个列表的元素，则结果为 1，相反则为－1，元素完全相同则结果为 0，类似于＞、＜、＝＝等关系运算符。

例如：

```
>>> cmp([1,2,5],[1,2,3])
1
>>> cmp([1,2,3],[1,2,3])
0
>>> cmp([123,'Bsaic'],[ 123,'Python'])
-1
```

在 Python 3.x 中，不再支持 cmp()函数，可以直接使用关系运算符来比较数值或列表。

例如：

```
>>> [123,'Bsaic']> [ 123,'Python']
False
>>> [1,2,3]== [1,2,3]
True
```

2. len()

格式：len(列表)

功能：返回列表中的元素个数。

例如：

```
>>> a_list= ['physics','chemistry',2017,2.5,[0.5,3]]
>>> len(a_list)
5
```

```
>>> len([1,2.0,'hello'])
3
```

3. max()和 min()

格式：max(列表),min(列表)

功能：返回列表中的最大或最小元素。要求所有元素之间可以进行大小比较。

例如：

```
>>> a_list= ['123','xyz','zara','abc']
>>> max(a_list)
'zara'
>>> min(a_list)
'123'
```

4. sum()

格式：sum(列表)

功能：对数值型列表的元素进行求和运算,对非数值型列表运算则出错。

例如：

```
>>> a_list= [23,59,-1,2.5,39]
>>> sum(a_list)
122.5
>>> b_list= ['123','xyz','zara','abc']
>>> sum(b_list)
Traceback (most recent call last):
  File "<pyshell# 11>",line 1,in <module>
    sum(b_list)
TypeError: unsupported operand type(s) for+ : 'int' and 'str'
```

5. sorted()

格式：sorted(列表)

功能：对列表进行排序,默认是按照升序排序。该方法不会改变原列表的顺序。

```
>>> a_list= [80,48,35,95,98,65,99,95,18,71]
>>> sorted(a_list)
[18,35,48,65,71,80,95,95,98,99]
>>> a_list                              #输出 a_list 列表,该列表原来的顺序并没有改变
[80,48,35,95,98,65,99,95,18,71]
```

如果需要进行降序排序,在 sorted()函数的列表参数后面增加一个 reverse 参数,其值等于 True 则表示降序排序,等于 False 则表示升序排序。例如：

```
>>> a_list= [80,48,35,95,98,65,99,95,18,71]
>>> sorted(a_list,reverse= True)
[99,98,95,95,80,71,65,48,35,18]
>>> sorted(a_list,reverse= False)
[18,35,48,65,71,80,95,95,98,99]
```

6. sort()

格式：list.sort()

功能：对列表进行排序,排序后的新列表会覆盖原列表,默认为升序排序。

```
>>> a_list= [80,48,35,95,98,65,99,95,18,71]
>>> a_list.sort()
>>> a_list                                  #输出 a_list 列表,该列表原来的顺序被改变了
[18,35,48,65,71,80,95,95,98,99]
```

如果需要进行降序排序,在 sort()方法中增加一个 reverse 参数,其值等于 True 表示则降序排序,等于 False 则表示升序排序。例如：

```
>>> a_list= [80,48,35,95,98,65,99,95,18,71]
>>> a_list.sort(reverse= True)
>>> a_list
[99,98,95,95,80,71,65,48,35,18]
>>> a_list.sort(reverse= False)
>>> a_list
[18,35,48,65,71,80,95,95,98,99]
```

7. reverse()

格式：list.reverse()

功能：对 list 列表中的元素进行翻转存放,不会对原列表进行排序。

```
>>> a_list= [80,48,35,95,98,65,99,95,18,71]
>>> a_list.reverse()
>>> a_list
[71,18,95,99,65,98,95,35,48,80]
```

列表基本操作及常用函数总结如表 6.1 所示。

表 6.1　列表基本操作及常用函数

方　法	功　　能
list. append(obj)	在列表末尾添加新的对象
list. extend(seq)	在列表末尾一次性追加另一个序列中的多个值
list. insert(index, obj)	将对象插入列表
list. index(obj)	从列表中找出某个值第一个匹配项的索引位置

续表

方　法	功　　能
list. count(obj)	统计某个元素在列表中出现的次数
list. remove(obj):	移除列表中某个值的第一个匹配项
list. pop(obj=list[-1])	移除列表中的一个元素(默认最后一个元素),并且返回该元素的值
sort()	对原列表进行排序
reverse()	反向存放列表元素
cmp(list1, list2)	比较两个列表的元素
len(list)	求列表元素个数
max(list)	返回列表元素的最大值
min(list)	返回列表元素的最小值
list(seq)	将元组转换为列表
sum(list)	对数值型列表元素求和

6.1.3 列表应用举例

【例 6.1】 从键盘上输入一批数据,对这些数据进行逆置,最后按照逆置后的结果输出。

分析:将输入的数据存放在列表中,将列表的所有元素镜像对调,即第一个与最后一个对调,第二个与倒数第二个对调,以此类推。

程序如下:

```
b_list=input("请输入数据:")
a_list=[]
for i in b_list.split(','):
              a_list.append(i)
print("逆置前数据为:",a_list)
n=len(a_list)
for i in range(n//2):
    a_list[i],a_list[n-i-1]=a_list[n-i-1],a_list[i]
print("逆置后数据为:",a_list)
```

程序运行结果:

```
请输入数据:"Python",2017,98.5,7102,'program'
逆置前数据为:['"Python"', '2017', '98.5', '7102', "'program'"]
逆置后数据为:["'program'", '7102', '98.5', '2017', '"Python"']
```

【例 6.2】 编写程序,求出 1000 以内的所有完数,并按下面的格式输出其因子:

```
6 its factors are 1,2,3
```

分析:一个数如果恰好等于它的因子之和,这个数就称为完数。例如,6 就是一个完数,因为 6 的因子有 1、2、3,且 6=1+2+3。

题目的关键是求因子。对于 2～100 的数，对于任意的数 a，采用循环对 1～$a-1$ 进行检测，如果检测到是 a 的因数，则将该因数存放在列表中，并将其累加起来，如果因数之和正好和该数相等，则数 a 是完数。

程序如下：

```
m=1000
for a in range(2,m+1):
        s=0
        L1=[]
        for i in range(1,a):
            if a%i==0:
                s+=i
                L1.append(i)
                if s==a:
                   print("%d  its factors are: "%a,L1)
```

程序运行结果：

```
6  its factors are: [1,2,3]
24  its factors are: [1,2,3,4,6,8]
28  its factors are: [1,2,4,7,14]
496  its factors are: [1,2,4,8,16,31,62,124,248]
```

6.2　元　　组

与列表类似，元组(tuple)也是 Python 的重要序列结构，但元组属于不可变序列，其元素不可改变，即元组一旦创建，用任何方法都不能修改元素的值，如果确实需要修改，只能再创建一个新元组。

元组的定义形式与列表类似，区别在于定义元组时所有元素放在一对圆括号“(”和“)”中。例如：

```
(1,2,3,4,5)
('Python', 'C','HTML','Java','Perl ')
```

6.2.1　元组的基本操作

1. 元组的创建

使用赋值运算符“=”将一个元组赋值给变量即可创建元组对象。例如：

```
>>>a_tuple=('physics','chemistry',2017,2.5)
>>>b_tuple=(1,2,(3.0,'hello world!'))              #元组中嵌套了元组类型
>>>c_tuple=('wade',3.0,81,['bosh','haslem'])       #元组中嵌套了列表类型
>>>d_tuple=()                                      #创建一个空元组
```

如果要创建只包含一个元素的元组，只把元素放在圆括号里是不行的，这是因为圆括号既可以表示元组，又可以表示数学公式中的小括号，产生了歧义。在这种情况下，Python做出规定，按小括号进行计算。因此要创建只包含一个元素的元组，需要在元素后面加一个逗号“,”，而创建多个元素的元组时则没有这个规定。例如：

```
>>> x= (1)
>>> x
1
>>> y= (1,)
>>> y
(1,)
>>> z= (1,2)
>>> z
(1,2)
```

注意：Python在显示只有一个元素的元组时，也会加一个逗号“,”，以免误解成数学计算意义上的括号。

2. 读取元素

与列表相同，使用索引可以直接访问元组的元素，方法为：元组名[索引]。例如：

```
>>> a_tuple= ('physics','chemistry',2017,2.5)
>>> a_tuple[1]
'chemistry'
>>> a_tuple[-1]
2.5
>>> a_tuple[5]
Traceback (most recent call last):
  File "<pyshell#14>",line 1,in <module>
    a_tuple[5]
IndexError: tuple index out of range
```

3. 元组切片

元组也可以进行切片操作，方法与列表类似。对元组切片可以得到一个新元组。例如：

```
>>> a_tuple[1:3]
('chemistry',2017)
>>> a_tuple[::3]
('physics',2.5)
```

4. 检索元素

(1) 使用元组对象的index()方法可以获取指定元素首次出现的下标。例如:

```
>>> a_tuple.index(2017)
2
>>> a_tuple.index('physics',-3)
Traceback (most recent call last):
  File "<pyshell#24>",line 1,in <module>
    a_tuple.index('physics',-3)
ValueError: tuple.index(x): x not in tuple          #在指定范围中没有检索到元素,提示错误信息
```

(2) 使用元组对象的count()方法统计元组中指定元素出现的次数。例如:

```
>>> a_tuple.count(2017)
1
>>> a_tuple.count(1)
0
```

(3) 使用in运算符检索某个元素是否在该元组中。如果元素在元组中,返回True,否则返回False。

```
>>> 'chemistry' in a_tuple
True
>>> 0.5 in a_tuple
False
```

5. 删除元组

使用del语句删除元组,删除之后对象就不存在了,再次访问会出错。例如:

```
>>> del a_tuple
>>> a_tuple
Traceback (most recent call last):
  File "<pyshell#30>",line 1,in <module>
    a_tuple
NameError: name 'a_tuple' is not defined
```

6.2.2 列表与元组的区别及转换

1. 列表与元组的区别

列表和元组在定义和操作上有很多相似的地方,不同点在于列表是可变序列,可以修改列表中元素的值,也可以增加和删除列表元素,而元组是不可变序列,元组中的数据一旦定义就不允许通过任何方式改变。因此,元组没有append()、insert()和extend()方法,不能给元组添加元素,也没有remove()和pop()方法,也不支持对元组元素进行del操

作，不能从元组删除元素。

与列表相比，元组具有以下优点。

(1) 元组的处理速度和访问速度比列表快。如果定义了一系列常量值，主要对其进行遍历或者类似用途，而不需要对其元素进行修改，这种情况一般使用元组。可以认为元组对不需要修改的数据进行了“写保护”，可以使代码更安全。

(2) 作为不可变序列，元组(包含数值、字符串和其他元组的不可变数据)可用作字典的键，而列表不可以充当字典的键，因为列表是可变的。

2. 列表与元组的转换

列表可以转换成元组，元组也可以转换成列表。内置函数 tuple()可以接收一个列表作为参数，返回包含同样元素的元组；list()可以接收一个元组作为参数，返回包含同样元素的列表。例如：

```
>>>a_list=['physics','chemistry',2017,2.5,[0.5,3]]
>>>tuple(a_list)
('physics','chemistry',2017,2.5,[0.5,3])
>>>type(a_list)                          #查看调用 tuple()函数之后 a_list 的类型
<class 'list'>                           #a_list 类型并没有改变
>>>b_tuple=(1,2,(3.0,'hello world!'))
>>>list(b_tuple)
[1,2,(3.0,'hello world!')]
>>>type(b_tuple)                         #查看调用 list()函数之后 b_tuple 的类型
<class 'tuple'>                          #b_tuple 类型并没有改变
```

从效果看，tuple()函数可以看作是在冻结列表使其不可变，而 list()函数是在融化元组使其可变。

6.2.3 元组应用

元组中元素的值不可改变，但元组中可变序列的元素的值可以改变。

可以利用元组来一次性给多个变量赋值。例如：

```
>>>v = ('a',2,True)
>>> (x,y,z)=v
>>>v = ('Python',2,True)
>>> (x,y,z)=v
>>>x
'Python'
>>>y
```



```
2
>>>z
True
```

6.3　字　符　串

Python 中的字符串是一个有序的字符集合,用于存储或表示基于文本的信息。它不仅能保存文本,而且能保存非打印字符或二进制数据。

Python 中的字符串用一对单引号(')或双引号(")引起来。例如:

```
>>> 'Python'
'Python'
>>> "Python Program"
'Python Program'
```

6.3.1　三重引号字符串

Python 中有一种特殊的字符串,用三重引号表示。如果字符串占据了几行,但却想让 Python 保留输入时使用的准确格式,例如行与行之间的回车符、引号、制表符或者其他信息都保存下来,则可以使用三重引号——字符串——以三个单引号或三个双引号开头,并且以三个同类型的引号结束。采用这种方式,可以将整个段落作为单个字符串进行处理。例如:

```
>>> '''Python is an "object- oriented"
open- source programming language'''
'Python is an "object- oriented"\n open- source programming language'
```

6.3.2　字符串基本操作

1. 字符串创建

使用赋值运算符“=”将一个字符串赋值给变量即可创建字符串对象。例如:

```
>>> str1= "Hello"
>>> str1
>>> str2= 'Program \n\'Python\''                #将包含有转义字符的字符串赋给变量
>>> str2
"Program \n'Python'"
```

2. 字符串元素读取

与列表相同,使用索引可以直接访问字符串中的元素,方法为:字符名[索引]。例如:

```
>>> str1[0]
'H'
>>> str1[- 1]
```

```
'o'
```

3. 字符串分片

字符串的分片就是从字符串中分离出部分字符，操作方法与列表相同，即采取“字符名[开始索引:结束索引:步长]”的方法。例如：

```
>>> str="Python Program"
>>> str[0:5:2]                #从第0个字符开始到第4个字符结束，每隔一个取一个字符
'Pto'
>>> str[:]                    #取出str字符本身
'Python Program'
>>> str[-1:-20]               #从-1开始，到-20结束，步长为1
''                            #结果为空串
>>> str[-1:-20:-1]            #将字符串由后向前逆向读取
'margorP nohtyP'
```

4. 连接

字符串连接运算可使用运算符“+”，将两个字符串对象连接起来，得到一个新的字符串对象。例如：

```
>>> "Hello"+"World"
'HelloWorld'
>>> "P"+"y"+"t"+"h"+"o"+"n"+"Program"
'PythonProgram'
```

将字符串和数值类型数据进行连接时，需要使用str()函数将数值数据转换成字符串，然后再进行连接运算。例如：

```
>>> "Python"+str(3)
'Python3'
```

5. 重复

字符串重复操作使用运算符“*”，构建一个由字符串自身重复连接而成的字符串对象。例如：

```
>>> "Hello"*3
'HelloHelloHello'
>>> 3*"Hello World!"
'Hello World!Hello World!Hello World!'
```

6. 关系运算

与数值类型数据一样，字符串也能进行关系运算，但关系运算的意义和在整型数据上

使用时略有不同。

(1) 单字符字符串的比较。单个字符字符串的比较是按照字符的ASCII码值大小进行比较。例如：

```
>>> "a"=="a"
True
>>> "a"=="A"
False
>>> "0">"1"
False
```

(2) 多字符字符串的比较。当字符串中的字符多于一个时，比较的过程仍是基于字符的ASCII码值的概念进行的。比较的过程是并行地检查两个字符串中位于同一位置的字符，然后向前推进，直到找到两个不同的字符为止。

① 从两个字符串中索引为0的位置开始比较。

② 比较位于当前位置的两个单字符。

如果两个字符相等，则两个字符串的当前索引加1，回到步骤②。

如果两个字符不相等，返回这两个字符的比较结果，作为字符串比较的结果。

③ 如果两个字符串比较到一个字符串结束时，对应位置的字符都相等，则较长的字符串更大。

例如：

```
>>> "abc"<"abd"
True
>>> "abc">"abcd"
False
>>> "abc"<"cde"
True
>>> ""<"0"
True
```

注意：空字符串(" ")比其他字符串都小，因为它的长度为0。

7. 成员运算

字符串使用in或not in运算符判断一个字符串是否属于另一个字符串。其一般形式为：

```
字符串1 [not] in 字符串2
```

其返回值为True或False。例如：

```
>>> "ab" in "aabb"
True
```

```
>>> "abc" in "aabbcc"
False
>>> "a" not in "abc"
False
```

6.3.3 字符串的常用方法

1. 子串查找

子串查找就是在主串中查找子串，如果找到则返回子串在主串中的位置，找不到则返回－1。Python 提供了 find()方法进行查找，其一般形式为：

```
str.find(substr,[start,[,end]])
```

其中，substr 是要查找的子串，start 和 end 是可选项，分别表示查找的开始位置和结束位置。例如：

```
>>> s1="beijing xi'an tianjin beijing chongqing"
>>> s1.find("beijing")
0
>>> s1.find("beijing",3)
22
>>> s1.find("beijing",3,20)
-1
```

2. 字符串替换

字符串替换 replace()方法的一般形式为：

```
str.replace(old,new[,max])
```

其中，old 是要进行更换的旧字符串，new 是用于替换 old 子字符串的新字符串，max 是可选项。该方法的功能是把字符串中的 old(旧字符串)替换成 new(新字符串)，如果指定了第三个参数 max，则替换不超过 max 次。例如：

```
>>> s2="this is string example. this is string example."
>>> s2.replace("is","was")                    # s2中所有的 is都替换成 was
'thwas was string example. thwas was string example.'
>>> s2="this is string example. this is string example."
>>> s2.replace("is","was",2)                  # s2中前面两个 is替换成 was
'thwas was string example. this is string example.'
```

3. 字符串分离

字符串分离是将一个字符串分离成多个子串组成的列表。Python 提供了 split()实现字符串的分离，其一般形式为：

```
str.split([sep])
```

其中，sep 表示分隔符，默认以空格作为分隔符。若参数中没有分隔符，则把整个字符串作为列表的一个元素，当有参数时，以该参数进行分隔。

```
>>> s3="beijing,xi'an,tianjin,beijing,chongqing"
>>> s3.split(',')                                  #以逗号作为分隔符
['beijing',"xi'an",'tianjin','beijing','chongqing']
>>> s3.split('a')                                  #以字符 a作为分隔符
>>> s3.split()                                     #没有分隔符,整个字符串作为列表的一个元素
["beijing,xi'an,tianjin,beijing,chongqing"]
```

4. 字符串连接

字符串连接是将列表、元组中的元素以指定的字符(分隔符)连接起来生成一个新的字符串，使用 join()方法实现，其一般形式为：

```
sep.join(sequence)
```

其中，sep 表示分隔符，可以为空；sequence 是要连接的元素序列。功能是以 sep 作为分隔符，将 sequence 所有的元素合并成一个新的字符串并返回该字符串。例如：

```
>>> s4=["beijing","xi'an","tianjin","chongqing"]
>>> sep="-->"
>>> str=sep.join(s4)                               #连接列表元素
>>> str                                            #输出连接结果
"beijing-->xi'an-->tianjin-->chongqing"
>>> s5=("Hello","World")
>>> sep=""
>>> sep.join(s5)                                   #连接元组元素
'HelloWorld'
```

字符串常用方法总结如表 6.2 所示。

表 6.2　字符串常用方法

方　　法	功　　能
str.find(substr,[start,[,end]])	定位子串 substr 在 str 中第一次出现的位置
str.replace(old,new(,max))	用字符串 new 替代 str 中的 old
str.split([sep])	以 sep 为分隔符，把 str 分隔成一个列表
sep.join(sequence)	把 sequence 的元素用连接符 seq 连接起来
str.count(substr,[start,[,end]])	统计 str 中有多少个 substr
str.strip()	去掉 str 中两端空格
str.lstrip()	去掉 str 中左边空格

续表

方　　法	功　　能
str.rstrip()	删除 str 中右边空格
str.strip([chars])	删除 str 中两端字符串 chars
str.isalpha()	判断 str 是否全是字母
str.isdigit()	判断 str 是否全是数字
str.isupper()	判断 str 是否全是大写字母
str.islower()	判断 str 是否全是小写字母
str.lower()	转换 str 中所有大写字母为小写
str.upper()	转换 str 中所有小写字母为大写
str.swapcase()	将 str 中的大小写字母互换
str.capitalize()	将字符串 str 中第一个字母变成大写,其他字母变小写

6.3.4 字符串应用举例

【例 6.3】 从键盘输入五个英文单词,输出其中以元音字母开头的单词。

分析:首先将所有的元音字母存放在字符串 str 中,然后循环地输入 10 个英文单词,并将这些单词存放在列表中。从列表中一一取出这些单词,采用分片的方法提取出单词的首字母,遍历存放元音的字符串 str,判断单词的首字母是否在 str 中。

程序如下:

```
str="AEIOUaeiou"
a_list=[]
for i in range(0,5):
    word=input("请输入一个英文单词:")
    a_list.append(word)
print("输入的 5个英文单词是:",a_list)
print("首字母是元音的英文单词有:")
for i in range(0,5):
     for ch in str:
          if a_list[i][0]==ch:
              print(a_list[i])
              break
```

程序运行结果:

```
请输入一个英文单词:china
请输入一个英文单词:program
请输入一个英文单词:Egg
请输入一个英文单词:apple
请输入一个英文单词:software
输入的五个英文单词是:['china','program','Egg','apple','software']
首字母是元音的英文单词有:
Egg
apple
```

【例6.4】 输入一段字符,统计其中单词的个数,单词之间用空格分隔。

分析:按照题意,连续的一段不含空格类字符的字符串就是单词。将连续的若干个空格看作出现一次空格,那么单词的个数可以由空格出现的次数(连续的若干个空格看作一次空格,一行开头的空格不统计)来决定。如果当前字符是非空格类字符,而它的前一个字符是空格,则可看作是新单词开始,累积单词个数的变量加1;如果当前字符是非空格类字符,而它的前一个字符也是非空各类字符,则可看作是旧单词的继续,累积单词个数的变量保持不变。

程序如下:

```
str=input("请输入一串字符:")
flag=0
count=0

for c in str:
    if c==" ":
        flag=0
    else:
        if flag==0:
            flag=1
            count=count+1
print("共有%d个单词"%count)
```

程序运行结果:

```
请输入一串字符:Python is an object-oriented  programming language often used for rapid
application  development
共有12个单词
```

【例6.5】 输入一行字符,分别统计出其中英文字母、空格、数字和其他字符的个数。

分析:首先输入一个字符串,根据字符串中每个字符的ASCII码值判断其类型。数字0~9对应的码值为48~57,大写字母A~Z对应的码值为65~90,小写字母a~z对应的码值为97~122。使用ord()函数将字符转换为ASCII码表上对应的数值。可以采用先找出各类型的字符,放到不同列表中,再分别计算列表的长度。

程序如下:

```
a_list=list(input('请输入一行字符:'))
letter=[]
space=[]
number = []
other = []

for i in range(len(a_list)):
    if ord(a_list[i]) in range(65,91) or ord(a_list[i]) in range(97,123):
        letter.append(a_list[i])
```

```
        elifa_list[i] == ' ':
            space.append(' ')
        elif ord(a_list[i]) in range(48,58):
            number.append(a_list[i])
        else:
            other.append(a_list[i])

print('英文字母个数：%s' % len(letter))
print('空格个数：%s' % len(apace))
print('数字个数：%s' % len(number))
print('其他字符个数：%s' % len(other))
```

程序运行结果：

```
请输入一行字符：Python 3.5.2 中文版
英文字母个数：6
空格个数：1
数字个数：3
其他字符个数：5
```

习　　题

1. 选择题

(1) 以下关于元组的描述正确的是(　　)。

A. 创建元组 tup：tup=()；　　B. 创建元组 tup：tup=(50)；

C. 元组中的元素允许被修改　　D. 元组中的元素允许被删除

(2) 以下语句的运行结果是(　　)。

```
>>Python = "  Python"
>>print("study"+ Python)
```

A. studyPython　　B. "study"Python

C. study Python　　D. 语法错误

2. 填空题

(1) 已知列表 a_list=['a','b','c','e','f','g']，写出实现以下功能的代码。

列出列表 a_list 的长度________________；

输出下标值为 3 的元素________________；

输出列表第 2 个及其后所有的元素________________；

增加元素'h'________________；

删除第 3 个元素________________。

(2) 按要求转换变量。

将字符串 str="python"转换为列表________________；

将字符串 str="python"转换为元组________________；

将列表 a_list=["python","Java","C"]转换为元组________________；

将元组 tup=("python","Java", "C")转换为列表________________。

(3) 已知字符串 str1, str2,判断 str1 是否是 str2 的一部分________________。

(4) 写出以下程序的运行结果________________。

```
def func(s,i,j):
    if i<j:
        func(s,i+1,j-1)
        s[i],s[j]=s[j],s[i]
def main():
    a=[10,6,23,-90,0,3]
    func(a,0,len(a)-1)
    for i in range(6):
        print (a[i])
main()
```

(5) 已知变量 str="abc:efg",写出实现以下功能的代码。

删除变量 str 两边的空格________________；

判断变量 str 是否以"ab"开始________________；

判断变量 str 是否以"g"结尾________________；

将变量 str 对应值中"e"替换为"x"________________；

输出变量 str 对应值的后 2 个字符________________。

3. 从键盘输入 10 个学生的成绩并存储在列表中,求成绩最高者的序号和成绩。

4. 编写程序,生成包含 20 个元素的随机数列表,将前 10 个元素升序排序,后 10 个元素降序排序,并输出结果。

5. 输入 10 名学生的成绩,进行优、良、中、及格和不及格的统计。

6. 将输入的字符串大小写进行转换并输出,例如,输入"aBc",输出"AbC"。

第 7 章 字典与集合

第 6 章讲到列表、元组和字符串都属于序列类型，序列元素之间有先后次序，可以通过位置编号的索引来访问序列的数据元素。本章介绍的字典与集合是 Python 中唯一内建的映射类型。字典中的值没有确定的顺序，这些值都存储在特定的“键”中。集合中的元素也是无序的，是由不重复元组组成的，类似于数学中集合的概念。

7.1 字 典

字典(dictionary)是 Python 语言中唯一的映射类型。这种映射类型由键(key)和值(value)组成，是键值对的无序可变序列。

定义字典时，每个元组的键和值用冒号分隔，相邻元素之间用逗号分隔，所有的元组放在一对大括号“{”和“}”中。字典中的键可以是 Python 中任意不可变类型，例如整数、实数、复数、字符串、元组等。键不能重复，而值可以重复。一个键只能对应一个值，但多个键可以对应相同的值。例如：

```
{1001: 'Alice',1002: 'Tom',1003: 'Emily'}
{(1,2,3): 'A',65.5,'B'}
{'Alice':95,'Beth':82,'Tom':65.5,'Emily':95}
```

7.1.1 字典常用操作

1. 字典的创建

(1) 使用“＝”将一个字典赋给一个变量即可创建一个字典变量。例如：

```
>>>a_dict={'Alice':95,'Beth':82,'Tom':65.5,'Emily':95}
>>>a_dict
{'Emily': 95,'Tom': 65.5,'Alice': 95,'Beth': 82}
```

也可创建一个空字典。例如：

```
>>>b_dict={}
>>>b_dict
{}
```

(2) 使用内建函数 dict()，通过其他映射(例如其他字典)或者(键，值)这样的序列对建立字典。例如：

```
#以映射函数的方式建立字典,zip()函数返回 tuple列表
>>>c_dict=dict(zip(['one','two','three'],[1,2,3]))
>>>c_dict
{'three': 3,'one': 1,'two': 2}
>>>d_dict =dict(one =1,two =2,three =3)            #以键值对方式建立字典
>>>d_dict
{'three': 3,'one': 1,'two': 2}
>>>e_dict=dict([('one',1),('two',2),('three',3)])  #以键值对形式的列表建立字典
>>>e_dict
{'three': 3,'one': 1,'two': 2}
>>>f_dict=dict((('one',1),('two',2),('three',3)))  #以键值对形式的元组建立字典
>>>f_dict
{'three': 3,'one': 1,'two': 2}
>>>g_dict=dict()                                   #创建空字典
>>>g_dict
{}
```

(3) 通过内建函数 fromkeys()来创建字典。

fromkeys()的一般形式为:

```
dict.fromkeys(seq[, value]))
```

其中,seq 表示字典键值列表;value 为可选参数,用于设置键序列(seq)的值。

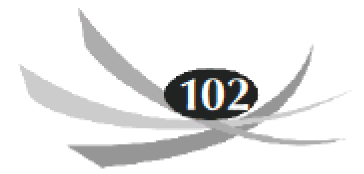
例如:

```
>>>h_dict={}.fromkeys((1,2,3),student)             #指定 value值为'student'
>>>h_dict
{1: 'student',2: 'student',3: 'student'}
#以给参数为键,不指定 value值,创建 value值为空的字典
>>>i_dict={}.fromkeys((1,2,3))
>>>i_dict
{1: None,2: None,3: None}
>>>j_dict={}.fromkeys(())                          #创建空字典
>>>j_dict
{}
```

2. 字典元素的读取

(1) 与列表和元组类似,可以使用下标的方式来访问字典中的元素,字典的下标是键,若使用的键不存在,则提示异常错误。例如:

```
>>>a_dict={'Alice':95,'Beth':82,'Tom':65.5,'Emily':95}
>>>a_dict['Tom']
65.5
>>>a_dict[95]
Traceback (most recent call last):
```




```
  File "<pyshell#32>",line 1,in <module>
    a_dict[95]
KeyError: 95
```

（2）使用字典对象的 get()方法获取指定键对应的值。get()方法的一般形式为：

```
dict.get(key, default=None)
```

其中，key 是指在字典中要查找的键，default 是指指定的键不存在时返回的值。该方法相当于一条 if…else 语句，如果参数 key 在字典中则返回 key 对应的 value 值，字典将返回 dict[key]；如果参数 key 不在字典中则返回参数 default，如果没有指定 default，默认值为 None。例如：

```
>>>a_dict.get('Alice')
95
>>>a_dict.get('a','address')              #键'a'在字典中不存在,返回指定的值
                                           #'address'
'address'
>>>a_dict.get('a')
>>>print(a_dict.get('a'))                 #键'a'在字典中不存在,没有指定值,返回默认的 None
None
```

3. 字典元素的添加与修改

（1）字典没有预定义大小的限制，可以随时向字典添加新的键值对，或者修改现有键所关联的值。添加和修改的方法相同，都是使用“字典变量名[键名]＝键值”的形式。区分究竟是添加还是修改，需要看键名与字典中的键名是否有重复。若该键存在，则表示修改该键的值；若不存在，则表示添加一个新的键值对，也就是添加一个新的元素。例如：

```
>>>a_dict['Beth']=79                      #修改键为'Beth'的值
>>>a_dict
{'Alice': 95,'Beth': 79,'Emily': 95,'Tom': 65.5}
>>>a_dict['Eric']=98                      #增加元素,键为'Eric',值为 98
>>>a_dict
{'Alice': 95,'Eric': 98,'Beth': 79,'Emily': 95,'Tom': 65.5}
```

（2）使用字典对象的 update()方法将另一个字典的键值对一次性全部添加到当前字典对象，如果当前字典中存在着相同的键，则以另一个字典中的值为准对当前字典进行更新。例如：

```
>>>a_dict={'Alice': 95,'Beth': 79,'Emily': 95,'Tom': 65.5}
>>>b_dict={'Eric':98,'Tom':82}
>>>a_dict.update(b_dict)                  #使用 update()方法修改 a_dict 字典
>>>a_dict
{'Alice': 95,'Beth': 79,'Emily': 95,'Tom': 82,'Eric': 98}
```

4. 删除字典中的元素

(1) 使用 del 命令删除字典中指定键对应的元素。例如：

```
>>> del a_dict['Beth']                    #删除键为 'Beth'的元素
>>> a_dict
{'Alice': 95,'Emily': 95,'Tom': 82,'Eric': 98}
>>> del a_dict[82]                        #删除键为 32 的元素,若不存在则提示出错
Traceback (most recent call last):
  File "<pyshell# 56>",line 1,in <module>
    del a_dict[82]
KeyError: 82
```

(2) 使用字典对象的 pop()方法删除并返回指定键的元素。例如：

```
>>> a_dict.pop('Alice')
95
>>> a_dict
{Emily': 95,'Tom': 82,'Eric': 98}
```

(3) 使用字典对象的 popitem()方法删除字典元素。由于字典是无序的，popitem()实际删除的是一个随机的元素。例如：

```
>>> a_dict.popitem()
('Emily',95)
>>> a_dict
{'Tom': 82,'Eric': 98}
```

(4) 使用字典对象的 clear()方法删除字典的所有元素。例如：

```
>>> a_dict.clear()
>>> a_dict
{}
```

5. 删除字典

使用 del 命令删除字典。例如：

```
>>> del a_dict
>>> a_dict
Traceback (most recent call last):
  File "<pyshell# 68>",line 1,in <module>
    a_dict
NameError: name 'a_dict' is not defined
```

注意：使用 clear()方法删除了字典的所有元素之后，字典为空；使用 del 命令删除了

字典,该对象被删除,再次访问就会出错。

7.1.2 字典的遍历

结合 for 循环语句,字典的遍历有很多方式。

1. 遍历字典的关键字

使用字典的 keys()方法,以列表的方式返回字典的所有键。keys()方法的语法为 dict. keys()。

例如:

```
>>> a_dict= {'Alice': 95,'Beth': 79,'Emily': 95,'Tom': 65.5}
>>> a_dict.keys()
dict_keys(['Tom','Emily','Beth','Alice'])
```

2. 遍历字典的值

使用字典的 values()方法,以列表的方式返回字典的所有值。values ()方法的语法为 dict. values()。例如:

```
>>> a_dict.values()
dict_values([65.5,95,79,95])
```

3. 遍历字典元素

使用字典的 items()方法,以列表的方式返回字典的所有(键,值)元素。items()方法的语法为 dict. items()。例如:

```
>>> a_dict.items()
dict_items([('Tom',65.5),('Emily',95),('Beth',79),('Alice',95)])
```

字典方法总结如表 7.1 所示。

表 7.1 字典方法

方法	功能
dict(seq)	用(键,值)对的(或者映射和关键字参数)建立字典
get(key [,returnvalue])	返回 key 的值。若无 key 而指定了 returnvalue,则返回 returnvalue 值;若无此值则返回 None
has_key(key)	如果 key 存在于字典中,就返回 1(真);否则返回 0(假)
items()	返回一个由元组构成的列表,每个元组包含一个键值对
keys()	返回一个由字典所有键构成的列表
popitem()	删除任意键值对,并作为两个元素的元组返回。若字典为空,则返回 KeyError 异常
update(newDictionary)	将来自 newDictionary 的所有键值添加到当前字典,并覆盖同名键的值
values()	以列表的方式返回字典的所有值
clear()	从字典删除所有项

7.1.3　字典应用举例

【例 7.1】　将一个字典的键和值对调。

分析：对调就是将字典的键变为值，值变为键。遍历字典，得到原字典的键和值，将原来的键作为值，原来的值作为键名，采用“字典变量名[键名]＝值”方式，逐个添加字典元素。

程序如下：

```
a_dict={'a':1,'b':2,'c':3}
b_dict={}
for key in a_dict:
    b_dict[a_dict[key]]=key
print(b_dict)
```

程序运行结果：

```
{1: 'a', 2: 'b', 3: 'c'}
```

【例 7.2】　输入一串字符，统计其中单词出现的次数，单词之间用空格分隔开。

分析：采用字典数据结构来实现。如果某个单词出现在字典中，可以将单词（键）作为索引来访问它的值，并将它的关联值加 1；如果某个单词（键）不存在于字典中，使用赋值的方式创建键，并将它的关联值置为 1。

程序如下：

```
string=input("input string:")
string_list=string.split()
word_dict={}
for word in string_list:
    if word in word_dict:
        word_dict[word]+=1
    else:
        word_dict[word] =1
print(word_dict)
```

程序运行结果：

```
input string:to be or not to be
{'or': 1,'not': 1,'to': 2,'be': 2}
```

7.2　集　　合

集合（set）是一组对象的组合，是一个无序排列的、不重复的数据组合体。类似于数学中的集合，可对其进行交、并、差等运算。

7.2.1 集合的常用操作

1. 创建集合

(1) 用一对大括号将多个用逗号分隔的数据括起来。例如：

```
>>>a_set={0,1,2,3,4,5,6,7,8,9}
>>>a_set
{0,1,2,3,4,5,6,7,8,9}
```

(2) 使用集合对象的 set()方法创建集合，该方法可以将列表、元组、字符串等类型的数据转换成集合类型的数据。例如：

```
#将列表类型的数据转换成集合类型
>>>b_set=set(['physics','chemistry',2017,2.5])
>>>b_set
{2017,2.5,'chemistry','physics'}
#将元组类型的数据转换成集合类型
>>>c_set=set(('Python','C','HTML','Java','Perl '))
>>>c_set
{'Java','HTML','C','Python','Perl '}
#将字符串类型的数据转换成集合类型
>>>d_set=set('Python')
>>>d_set
{'y','o','t','h','n','P'}
```

(3) 使用集合对象的 frozenset()方法创建一个冻结的集合，即该集合不能再添加或删除任何集合里的元素。它与 set()方法创建的集合区别是：set()方法可以添加或删除元素，而 frozenset()方法则不行；frozenset()方法可以作为字典的 key，也可以作为其他集合的元素，而 set()方法不可以。例如：

```
>>>e_set=frozenset('a')                    #正确
>>>e_set
>>>a_dict={e_set:1,'b':2}
>>>a_dict
{frozenset({'a'}): 1,'b': 2}
>>>f_set=set('a')
>>>f_set
{'a'}
>>>b_dict={f_set:1,'b':2}                  #错误
Traceback (most recent call last):
  File "<pyshell#9>",line 1,in <module>
    b_dict={f_set:1,'b':2}
TypeError: unhashable type: 'set'
```

注意：在集合中不允许有相同元素，如果在创建集合时有重复元素，Python 会自动删除重复的元素。例如：

```
>>> g_set= {0,0,0,0,1,1,1,3,4,5,5,5}
>>> g_set
{0,1,3,4,5}
```

2. 访问集合

由于集合本身是无序的，所以不能为集合创建索引或进行切片操作，只能使用 in、not in 或者循环遍历来访问或判断集合元素。例如：

```
>>> b_set= set(['physics','chemistry',2017,2.5])
>>> b_set
{'chemistry',2017,2.5,'physics'}
>>> 2.5 in b_set
True
>>> 2 in b_set
False
>>> for i in b_set:print(i,end= ' ')
chemistry 2017 2.5 physics
```

3. 删除集合

使用 del 语句删除集合。例如：

```
>>> a_set= {0,1,2,3,4,5,6,7,8,9}
>>> a_set
{0,1,2,3,4,5,6,7,8,9}
>>> del a_set
>>> a_set
Traceback (most recent call last):
  File "<pyshell# 66>",line 1,in <module>
    a_set
NameError: name 'a_set' is not defined
```

4. 更新集合

使用以下内建方法来更新可变集合。

(1) 使用集合对象的 add()方法为集合添加元素。一般格式为：s. add(x)，其功能是在集合 s 中添加元素 x。例如：

```
>>> b_set.add('math')
>>> b_set
{'chemistry',2017,2.5,'math','physics'}
```

(2) 使用集合对象的 update()方法修改集合。一般格式为：s. update(s1,s2,…,sn)，其功能是用集合 s1,s2,…,sn 中的成员修改集合 s，s=s∪s1∪s2∪…∪sn。例如：

```
>>> s={'Python','C','C++'}
>>> s.update({1,2,3},{'Wade','Nash'},{0,1,2})
>>> s
{0,1,2,3,'Python','Wade','C++','Nash','C'}      #去除了重复的元素
```

5. 删除集合中的元素

(1) 使用集合对象的 remove()方法删除集合元素。一般格式为：s. remove(x)，其功能是从集合 s 中删除元素 x，若 x 不存在，则提示出错信息。例如：

```
>>> s={0,1,2,3,'Phthon','Wade','C++','Nash','C'}
>>> s.remove(0)
>>> s
{1,2,3,'Python','Wade','C++','Nash','C'}
>>> s.remove('Hello')
Traceback (most recent call last):
  File "<pyshell#45>",line 1,in <module>
    s.remove('Hello')
KeyError: 'Hello'
```

(2) 使用集合对象的 discard()方法删除集合元素。一般格式为：s. discard(x)，其功能是从集合 s 中删除元素 x，若 x 不存在，不提示出错信息。例如：

```
>>> s.discard('C')
>>> s
{1,2,3,'Python','Wade','C++','Nash'}
>>> s.discard('abc')
>>> s
{1,2,3,'Python','Wade','C++','Nash'}
```

(3) 使用集合对象的 pop()方法删除集合中任意一个元素并返回该集合。例如：

```
>>> s.pop()
1
>>> s
{2,3,'Python','Wade','C++','Nash'}
```

(4) 使用集合对象的 clear()方法删除集合的所有元素。例如：

```
>>> s.clear()
>>> s
set()                                   #空集合
```

7.2.2　集合常用运算

Python 提供的方法实现了典型的数学集合运算，支持一系列标准操作。

1. 交集

方法：s1&s2&…&sn，计算 s1，s2，…，sn 这 n 个集合的交集。例如：

```
>>> {0,1,2,3,4,5,7,8,9}&{0,2,4,6,8}
{8,0,2,4}
>>> {0,1,2,3,4,5,7,8,9}&{0,2,4,6,8}&{1,3,5,7,9}
set()                                          #交集为空集合
```

2. 并集

方法：s1|s2|…|sn，计算 s1，s2，…，sn 这 n 个集合的并集。例如：

```
>>> {0,1,2,3,4,5,7,8,9}|{0,2,4,6,8}
{0,1,2,3,4,5,6,7,8,9}
>>> {0,1,2,3,4,5}|{0,2,4,6,8}
{0,1,2,3,4,5,6,8}
```

3. 差集

方法：s1－s2－…－sn，计算 s1，s2，…，sn 这 n 个集合的差集。例如：

```
>>> {0,1,2,3,4,5,6,7,8,9}- {0,2,4,6,8}
{1,3,5,9,7}
>>> {0,1,2,3,4,5,6,7,8,9}- {0,2,4,6,8}- {2,3,4}
{1,5,9,7}
```

4. 对称差集

方法：s1^s2^…^sn，计算 s1，s2，…，sn 这 n 个集合的对称差集，即求所有集合的相异元素。例如：

```
>>> {0,1,2,3,4,5,6,7,8,9}^{0,2,4,6,8}
{1,3,5,7,9}
>>> {0,1,2,3,4,5,6,7,8,9}^{0,2,4,6,8}^{1,3,5,7,9}
set()
```

5. 集合的比较

（1）s1＝＝s2：判断 s1 和 s2 集合是否相等。如果 s1 和 s2 集合具有相同的元素，则返回 True；否则返回 False。例如：

```
>>> {1,2,3,4}== {4,3,2,1}
```

```
True
```

注意：判断两个集合是否相等，只需要判断其中的元素是否一致，与顺序无关。

(2) s1！=s2：判断s1和s2集合是否不相等，如果s1和s2集合具有不同的元素，则返回True；否则返回False。例如：

```
>>> {1,2,3,4}!= {4,3,2,1}
False
>>> {1,2,3,4}!= {2,4,6,8}
True
```

(3) s1<s2：判断集合s1是否是集合s2的真子集。如果s1不等于s2，且s1中所有元素都是s2的元素，则返回True；否则返回False。例如：

```
>>> {1,2,3,4}< {4,3,2,1}
False
>>> {1,2,3,4}< {1,2,3,4,5}
True
```

(4) s1<=s2：判断集合s1是否是集合s2的子集。如果s1中所有元素都是s2的元素，则返回True；否则返回False。例如：

```
>>> {1,2,3,4}<= {1,2,3,4}
True
>>> {1,2,3,4}< {1,2,3,4,5}
True
```

(5) s1>s2：判断集合s1是否是集合s2的真超集。如果s1不等于s2，且s2中所有元素都是s1的元素，则返回True；否则返回False。例如：

```
>>> {1,2,3,4}> {4,3,2,1}
False
>>> {1,2,3,4}> {3,2,1}
True
```

(6) s1>=s2：判断集合s1是否是集合s2的超集。如果s2中所有元素都是s1的元素，则返回True；否则返回False。例如：

```
>>> {1,2,3,4}>= {4,3,2,1}
True
>>> {1,2,3,4}>= {3,2,1}
True
```

适合于可变集合的方法如表7.2所示。

表 7.2　可变集合方法

方　　法	功　　能
s. update(t)	用 t 中的元素修改 s，即修改之后 s 中包含 s 和 t 的成员
s. add(obj)	在 s 集合中添加对象 obj
s. remove(obj)	从集合 s 中删除对象 obj，如果 obj 不是 s 中的元素，将触发 KeyError 错误
s. discard(obj)	如果 obj 是集合 s 中的元素，从集合 s 中删除对象 obj
s. pop()	删除集合 s 中的任意一个对象，并返回该对象
s. clear()	删除集合 s 中的所有元素

习　　题

1. 选择题

(1) 以下关于字典的描述错误的是(　　)。

A. 字典是一种可变容器，可存储任意类型对象

B. 每个键值对都用冒号(:)隔开，每个键值对之间用逗号(,)隔开

C. 键值对中，值必须唯一

D. 键值对中，键必须是不可变的

(2) 下列说法错误的是(　　)。

A. 除字典类型外，所有标准对象均可以用于布尔测试

B. 空字符串的布尔值是 False

C. 空列表对象的布尔值是 False

D. 值为 0 的任何数字对象的布尔值是 False

(3) 以下不能创建字典的语句是(　　)。

A. dict1={}　　　　B. dict2={3:5}

C. dict3={[1,2,3]: "uestc"}　　　　D. dict4={(1,2,3): "uestc"}

2. 已知字典 dict={"name":"Zhang","Address":"Shaanxi","Phone":"123556"}，编写代码实现以下功能。

(1) 分别输出 dict 所有的键(key)、值(value)。

(2) 输出 dict 的 Address 值。

(3) 修改 dict 的 Phone 值为"029-8888 8888"。

(4) 添加键值对"class":"Python"，并输出。

(5) 删除字典 dict 的 Address 键值对。

3. 编写购物车程序，购物车类型为列表类型，列表的每个元素为一个字典类型，字典键值包括"name","price"，使用函数实现如下功能。

(1) 创建购物车：键盘输入商品信息，并输出商品列表。例如，输入：

鼠标 66
键盘 888
固态硬盘 599

购物车列表为

```
goods=[
{"name":"电脑","price":1999},
{"name":"鼠标","price":66},
{"name":"键盘","price":888},
{"name":"固态硬盘","price":599},
]
```

(2) 从键盘输入用户资产(如2000),按序号选择商品,加入购物车,若商品总额大于用户资产,提示用户余额不足,否则购买成功。

4. 设计一个字典,用户输入内容作为键,查找输出字典中对应的值,如果用户输入的键不存在,则输出“该键不存在!”。

5. 已知列表a_list=[11,22,33,44,55,66,77,88,99,90],将所有大于60的值保存至字典的第一个key的值中,将小于60的值保存至第二个key的值中,即{'k1':大于66的所有值;'k2':小于66的所有值}。

第 8 章 函数与模块

人们在求解某个复杂问题时，通常采用逐步分解、分而治之的方法，也就是将一个大问题分解成若干个比较容易求解的小问题，然后分别求解。同样，程序员在设计一个复杂的应用程序时，当编写的程序代码越来越多、越来越复杂，为了使程序更简洁、可读性更好、更易于维护，也是把整个程序划分成若干个功能较为单一的程序模块，然后分别予以实现，最后再把所有的程序模块像搭积木一样装配起来。这种在程序设计中分而治之的策略称为模块化程序设计方法。Python 语言通过函数来实现程序的模块化。利用函数可以化整为零，简化程序设计。

Python 还提供模块的方式组织程序单元，模块可以看作是一组函数的集合，一个模块可以包含若干个函数。

8.1 函数概述

函数是一组实现某一特定功能的语句集合，是可以重复调用、功能相对独立完整的程序段。可以把函数看成一个“黑盒子”，只要输入数据就能得到结果，而函数内部究竟是如何工作的，外部程序是不知道的，外部程序所知道的仅限于给函数输入什么，以及函数执行后输出什么。

1. 使用函数的优点

在编写程序时，使用函数具有明显的优点。

1）实现程序的模块化

当需要处理的问题比较复杂时，把一个大问题划分为若干个小问题，每一个小问题相对独立。不同的小问题，可以分别采用不同的方法加以处理，做到逐步求精。

2）减轻编程、维护的工作量

把程序中常用的一些计算或操作编写成通用的函数，以供随时调用，可以大大减少程序员的编码及维护的工作量。

2. 函数分类

在 Python 中，可以从不同的角度对函数进行分类。

1）从用户的使用角度

从用户的使用角度，函数可分为以下两种。

(1) 标准库函数,也称标准函数。这是由 Python 系统提供的,用户不必定义,只需在程序前面导入该函数原型所在的模块,就可以在程序中直接调用。在 Python 中,提供了很多库函数,可以方便用户使用。

(2) 用户自定义的函数。由用户按需要、遵循 Python 语言的语法规则自己编写的一段程序,用以实现特定的功能。

2) 从函数参数传递的角度

从函数参数传递的角度,函数可分为以下两种。

(1) 有参函数。在函数定义时带有参数的函数。在函数定义时的参数称为形式参数(简称形参),在相应的函数调用时也必须有参数,称为实际参数(简称实参)。在函数调用时,主调函数和被调函数之间通过参数进行数据传递。主调函数可以把实际参数的值传递被调函数的形式参数。

(2) 无参函数。在函数定义时没有形式参数的函数。在调用无参函数时,主调函数并不将数据传递给被调函数。

8.2 函数的定义与调用

8.2.1 函数定义

在 Python 中,函数定义基本函数的一般形式为:

```
def 函数名([形式参数表]):
    函数体
    [return 表达式]
```

函数定义时要注意以下几点。

(1) 采用 def 关键字进行函数的定义,不需要指定返回值的类型。

(2) 函数的参数可以是零个、一个或者多个,不需要指定参数类型,多个参数之间用逗号分隔。

(3) 参数括号后面的冒号":"必不可少。

(4) 函数体相对于 def 关键字必须保持一定的空格缩进。

(5) return 语句是可选的,它可以在函数体内任何地方出现,表示函数调用执行到此结束。如果没有 return 语句,会自动返回 None;如果有 return 语句,但是 return 后面没有表达式也返回 None。

(6) Python 还允许定义函数体为空的函数,其一般形式为:

```
def 函数名():
     pass
```

pass 语句什么都不做,用来作为占位符,即调用此函数时,什么工作也不做。空函数出现在程序中的主要目的为:在函数定义时,因函数的算法还未确定、暂时来不及编写或有待于完善和扩充功能等,未给出函数完整的定义。在程序开发过程中,通常先开发主要

函数，次要的函数或准备扩充程序功能的函数暂写成空函数，使程序在不完整的情况下能调试部分程序。

【例8.1】 定义函数，输出“Hello world!”。

程序如下：

```
def printHello():                #不带参数,没有返回值
    print("Hello world!")
```

【例8.2】 定义函数，求两个数的最大值。

程序如下：

```
def max(a,b):
    if a>b:
        return a
    else:
        return b
```

8.2.2 函数调用

在Python中通过函数调用来进行函数的控制转移和相互间数据的传递，并对被调函数进行展开执行。

1. 函数调用的一般形式

函数调用的一般形式为：

```
函数名([实际参数表])
```

函数调用时传递的参数是实参，实参可以是变量、常量或表达式。当实参个数超过一个时，用逗号分隔，实参和形参应在个数、类型和顺序上一一对应。对于无参函数，调用时实参表列为空，但()不能省略。

2. 函数调用的一般过程

函数调用的一般过程如下。

(1) 为所有形参分配内存单元，再将主调函数的实参传递给对应的形参。

(2) 转去执行被调用函数，为函数体内的变量分配内存单元，执行函数体内语句。

(3) 遇到return语句时，返回主调函数并带回返回值(无返回值的函数例外)，释放形参及被调用函数中各变量所占用的内存单元，返回到主调函数继续执行。若程序中无return语句，则执行完被调用函数后回到主调函数。

【例8.3】 编写函数，求三个数中的最大值。

程序如下：

```
defgetMax(a,b,c):
    if a>b:
        max=a
    else:
```

```
            max = b
        if(c>m):
            max = c
        return max

a,b,c=eval(input("input a,b,c:"))
n=getMax (a,b,c)
print("max=",n)
```

程序运行结果：

```
input a,b,c:10,43,23
max= 43
```

注意：在 Python 中不允许前向引用，即在函数定义之前，不允许调用该函数。例如有如下程序：

```
print(add(1,2))

def add(a,b):
    return a+b
```

程序运行结果：

```
Traceback (most recent call last):
  File "F:/python/add.py",line 1,in <module>
    print(add(1,2))
NameError: name 'add' is not defined
```

从给出的错误类型可以知道，名字为'add'的函数未进行定义。所以在任何时候调用函数，必须确保其定义在调用之前，否则运行将出错。

8.3 函数的参数及返回值

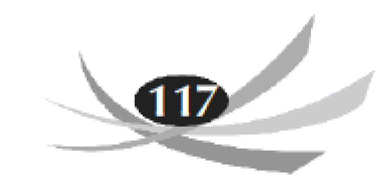
函数作为一个数据处理的功能部件，是相对独立的。但在一个程序中，各函数要共同完成一个总的任务，所以函数之间必然存在数据传递。函数间的数据传递包括如下两个方面。

(1) 数据从主调函数传递给被调函数(通过函数的参数实现)。

(2) 数据从被调函数返回到主调函数(通过函数的返回值实现)。

8.3.1 形式参数和实际参数

在函数定义的首部，函数名后括号中变量称为形式参数，简称形参。形参的个数可以有多个，多个形参之间用逗号隔开。与形参相对应，当一个函数被调用的时候，在被调用处给出对应的参数，这些参数称为实际参数，简称实参。



根据实参传递给形参值的不同，通常有值传递和地址传递两种方式。

1. 值传递方式

所谓值传递方式是指函数调用时，为形参分配存储单元，并将实参的值复制到形参；函数调用结束，形参所占内存单元被释放，值消失。

其特点是：形参和实参各占不同的内存单元，函数中对形参值的改变不会改变实参的值，这就是函数参数的单向传递规则。

【例 8.4】　函数参数的值传递方式。

程序如下：

```
def swap(a,b):
    a,b=b,a
    print("a=",a,"b=",b)

x,y=eval(input("input x,y:"))
swap(x,y)
print("x=",x,"y=",y)
```

程序运行结果：

```
input x,y:3,5
a=5 b=3
x=3 y=5
```

在调用 swap(a,b)时，实参 x 的值传递给形参 a，实参 y 的值传递给形参 b，在函数中通过交换赋值，将 a 和 b 的值进行交换。从程序运行结果可以看出，形参 a 和 b 的值进行了交换，而实参 x 和 y 的值并没有交换。其函数参数值传递调用的过程如图 8.1 所示。

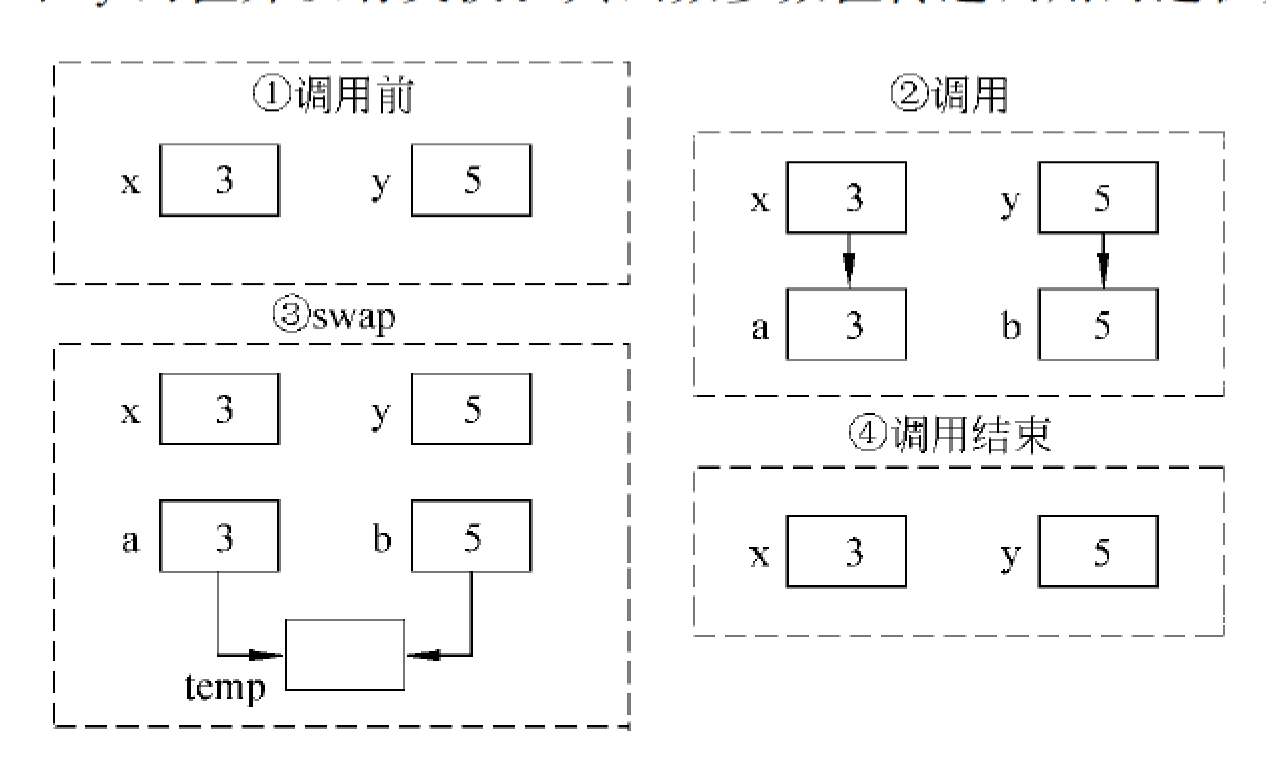


图 8.1　函数参数值传递方式

2. 地址传递方式

所谓地址传递方式是指在函数调用时，将实参数据的存储地址作为参数传递给形参。

其特点是：形参和实参占用同样的内存单元，函数中对形参值的改变也会改变实参的值。因此，函数参数的地址传递方式可以实现调用函数与被调用函数之间的双向数据传递。

Python 中将列表对象作为函数的参数，则向函数中传递的是列表的引用地址。

【例 8.5】 函数参数的地址传递方式。

程序如下：

```
def swap(a_list):
    a_list[0],a_list[1]=a_list[1],a_list[0]
    print("a_list[0]=",a_list[0],"a_list[1]=",a_list[1])


x_list=[3,5]
swap(x_list)
print("x_list[0]=",x_list[0],"x_list[1]=",x_list[1])
```

程序运行结果：

```
a_list[0]=5 a_list[1]=3
x_list[0]=5 x_list[1]=3
```

在调用 swap(a_list)时，将列表对象实参 x_list 的地址传递给形参 a_list，x_list 和 a_list 指向同一个内存单元，函数中当 a_list[0]和 a_list[1]进行数据交换时，也使 x_list[0]和 x_list[1]的值进行了交换。

8.3.2 函数的返回值

函数的返回值是指函数被调用、执行完后返回给主调函数的值。一个函数可以有返回值，也可以没有返回值。

返回语句的一般形式为：

```
return 表达式
```

功能：将表达式的值带回给主调函数。当执行完 return 语句时，程序的流程就退出被调函数，返回到主调函数的断点处。

(1) 在函数内可以根据需要有多条 return 语句，但执行到哪条 return 语句，相应的 return 语句就起作用，如例 8.2。

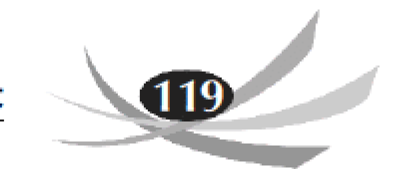
(2) 如果没有 return 语句，会自动返回 None；如果有 return 语句，但是 return 后面没有表达式也返回 None。例如：

```
def add(a,b):
    c=a+b


x=add(3,20)
print(x)
```

程序运行结果：

```
x=None
```



【例 8.6】　编写函数，判断一个数是否是素数。

分析：所谓素数是指仅能被 1 和自身整除的大于 1 的整数。

程序如下：

```
def isprime(n):
  for i in range(2,n):
      if(n% i==0):
          return 0
  return 1

m=int(input("请输入一个整数:"))
flag=isprime(m)
if(flag==1):
      print("% d是素数"% m)
else:
      print("% d不是素数"% m)
```

程序运行结果：

```
请输入一个整数:35
35 不是素数
```

再次运行程序，结果如下：

```
请输入一个整数:5
5 是素数
```

说明：isprime()函数是根据形参值是否为素数决定返回值，函数体最后将判断的结果由 return 语句返回给主调函数。

(3) 如果需要从函数中返回多个值时，可以使用元组作为返回值，来间接达到返回多个值的作用。

【例 8.7】　求一个数列中的最大值和最小值。

程序如下：

```
def getMaxMin(x):
    max=x[0]
    min=x[0]
    for i in range(0,len(x)):
        if max<x[i]:
            max=x[i]
        if min>x[i]:
            min=x[i]
    return (max,min)

a_list=[-1,28,-15,5,10]          #测试数据为列表类型
```

```
x,y=getMaxMin(a_list)
print("a_list=",a_list)
print("最大元素=",x,"最小元素=",y)

string="Hello"                  #测试数据为字符串
x,y=getMaxMin(string)
print("string=",string)
print("最大元素=",x,"最小元素=",y)
```

程序运行结果：

```
a_list=[-1,28,-15,5,10]
最大元素=28 最小元素=-15
string=Hello
最大元素=o 最小元素=H
```

说明：返回语句“return max,min”也可以写成“return (max,min)”，返回的是元组类型的数据，在测试程序中，分别赋给变量 x 和 y。

8.4 递归函数

在函数的执行过程中又直接或间接地调用该函数本身，这就是函数的递归调用。Python 中允许递归调用。在函数中直接调用函数本身称为直接递归调用。在函数中调用其他函数，其他函数又调用原函数，称为间接递归调用。函数的递归调用如图 8.2 所示。

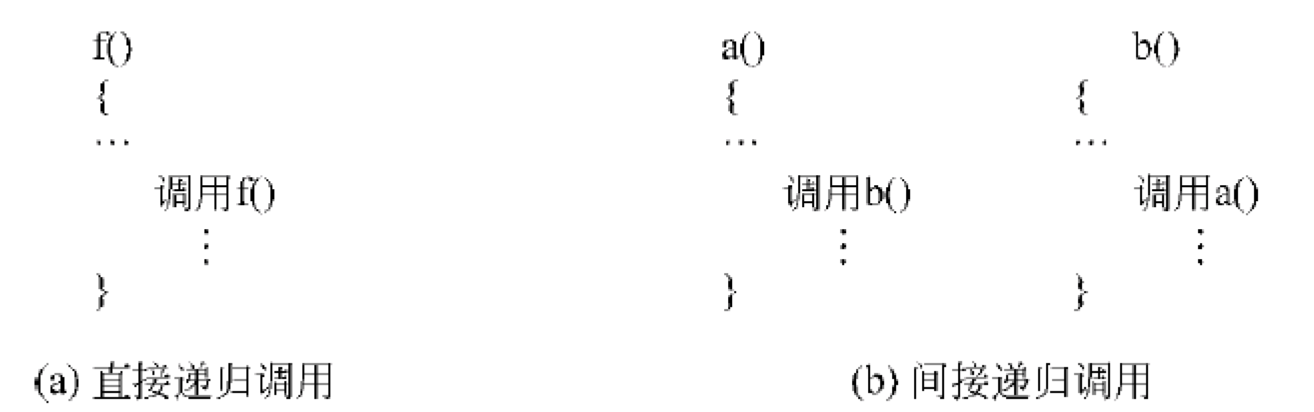


图 8.2 函数的递归调用

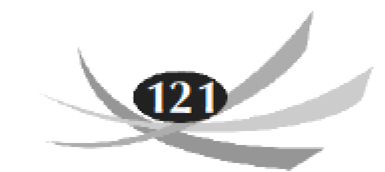
例如，求一个数 n 的阶乘：

$$n! = \begin{cases} 1 & \text{当 } n = 0 \text{ 时} \\ n*(n-1)! & \text{当 } n > 0 \text{ 时} \end{cases}$$

在求解 $n!$中使用了$(n-1)!$，即要计算出 $n!$，必须先求出$(n-1)!$，而要知道$(n-1)!$，必须先求出$(n-2)!$，以此类推，直到求出 $0!=1$ 为止。再以此为基础，返回来计算 $1!$，$2!$，…，$(n-1)!$，$n!$。这种算法称为递归算法。递归算法可以将复杂问题化简。显然，通过函数的递归调用可以实现递归算法。

递归算法具有以下两个基本特征。

(1) 递推归纳(递归体)。将问题转化成比原问题规模小的同类问题，归纳出一般递



推公式。问题规模往往需要用函数的参数来表示。

(2) 递归终止(递归出口)。当规模小到一定程度时应该结束递归调用,逐层返回。常用条件语句来控制何时结束递归。

【例 8.8】 用递归方法求 $n!$。

递推归纳:$n! \to (n-1)! \to (n-2)! \to \cdots \to 2! \to 1!$,得到递推公式 $n! = n * (n-1)!$。

递归终止条件:当 $n=0$ 时,$0!=1$。

程序如下:

```
def fac(n):
  if  n==0:
      f=1
  else:
      f=fac(n-1) * n;
  return f

n=int(input("please input n: "))
f=fac(n)
print("%d!=%d"%(n,f))
```

程序运行结果:

```
please input n: 4
4!=24
```

计算 4!时 fac()函数的递归调用过程如图 8.3 所示。

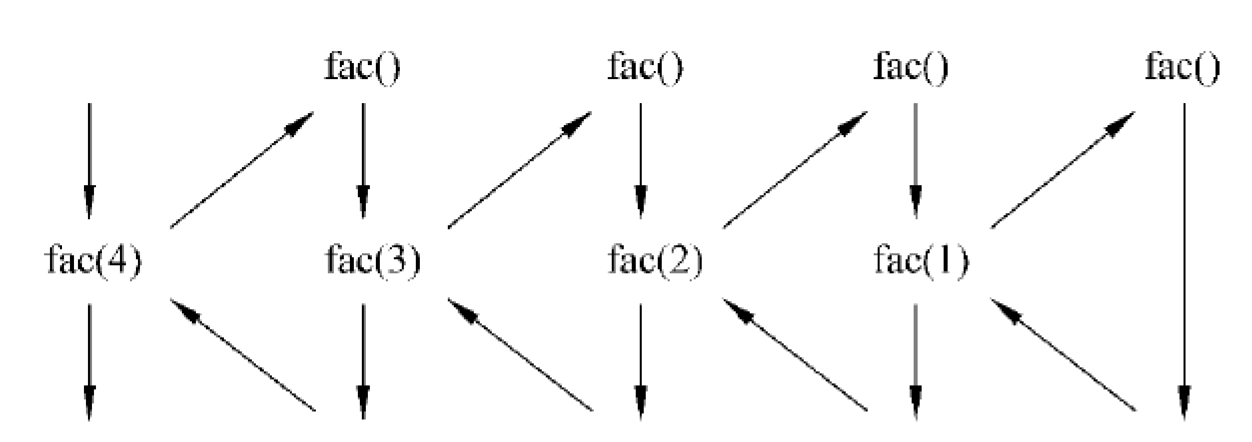


图 8.3　计算 4!时 fac()函数的递归调用过程

递归调用的执行分成两个阶段完成:第一阶段是逐层调用,调用的是函数自身;第二阶段是逐层返回,返回到调用该层的位置继续执行后续操作。

递归调用是多重嵌套调用的一种特殊情况,每层调用都要用堆栈保护主调层的现场和返回地址。调用的层数一般比较多,递归调用的层数称为递归的深度。

【例 8.9】 汉诺(Hanoi)塔问题。

假设有三个塔座,分别用 A、B、C 表示,在一个塔座(设为 A 塔)上有 64 个盘片,盘片大小不等,按大盘在下、小盘在上的顺序叠放着,如图 8.4 所示。现要借助于 B 塔,将这些盘片移到 C 塔,要求在移动的过程中,每个塔座上的盘片始终保持大盘在下、小盘在上的叠放方式,每次只能移动一个盘片。编程实现移动盘片的过程。

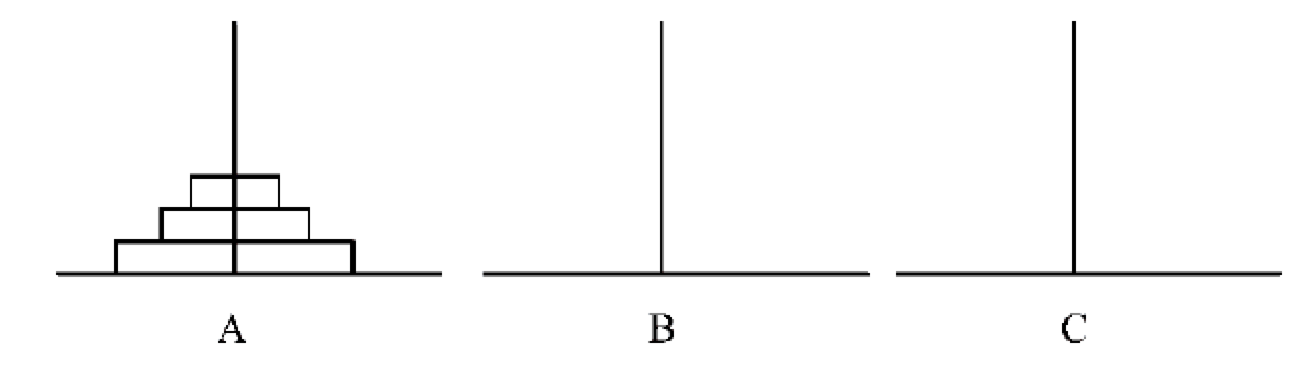


图 8.4 汉诺(Hanoi)塔问题

可以设想：只要能将除最下面的一个盘片外，其余的 63 个盘片从 A 塔借助于 C 塔移至 B 塔上，剩下的一片就可以直接移至 C 塔上。再将其余的 63 个盘片从 B 塔借助于 A 塔移至 C 塔上，问题就解决了。这样就把一个 64 个盘片的汉诺塔问题化简为 2 个 63 个盘片的汉诺塔问题，而每个 63 个盘片的汉诺塔问题又按同样的思路，可以简化为 2 个 62 个盘片的汉诺塔问题。继续递推，直到剩一个盘片时，可直接移动，递归结束。

编程实现：假设要将 n 个盘片按规定从 A 塔移至 C 塔，移动步骤可分为以下三步完成。

(1) 把 A 塔上的 n－1 个盘片借助 C 塔移至 B 塔。

(2) 把第 n 个盘片从 A 塔移至 C 塔。

(3) 把 B 塔上的 n－1 个盘片借助 A 塔移至 C 塔。

本例用函数 hanoi (n, x, y, z) 以递归算法实现，hanoi() 函数的形参为 n、x、y、z，分别存储盘片数、源塔、借用塔和目的塔。调用函数每调用一次，可以使盘片数减 1，当递归调用盘片数为 1 时结束递归。算法描述如下。

如果 n 等于 1，则将这一个盘片从 x 塔移至 z 塔，否则有：

(1) 递归调用 hanoi (n－1, x, z, y)，将 n－1 个盘片从 x 塔借助 z 塔移至 y 塔。

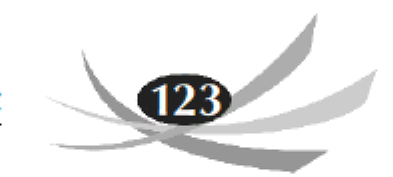
(2) 将 n 号盘片从 x 塔移至 z 塔。

(3) 递归调用 hanoi (n－1, y, x, z)，将 n－1 个盘片从 y 塔借助 x 塔移至 z 塔。

程序如下：

```
count=0
def hanoi(n,x,y,z):
    global count
if n==1:
    count+=1
    move(count,x,z)
else:
    hanoi(n-1,x,z,y);          #递归调用
    count+=1
    move(count,x,z)
    hanoi(n-1,y,x,z);          #递归调用

def move(n,x,y):
    print("step %d: Move disk from %c to %c"% (count,x,y))
```




```
m= int(input("Input the number of disks:"))
print("The steps to moving %d disks:"% m)
hanoi(m,'A','B','C')
```

程序运行结果：

```
Input the number of disks:3
Thesteps to moving 3 disks:
step 1: Move disk from A to C
step 2: Move disk from A to B
step 3: Move disk from C to B
step 4: Move disk from A to C
step 5: Move disk from B to A
step 6: Move disk from B to C
step 7: Move disk from A to C
```

n＝3 时函数的递归调用过程如图 8.5 所示。

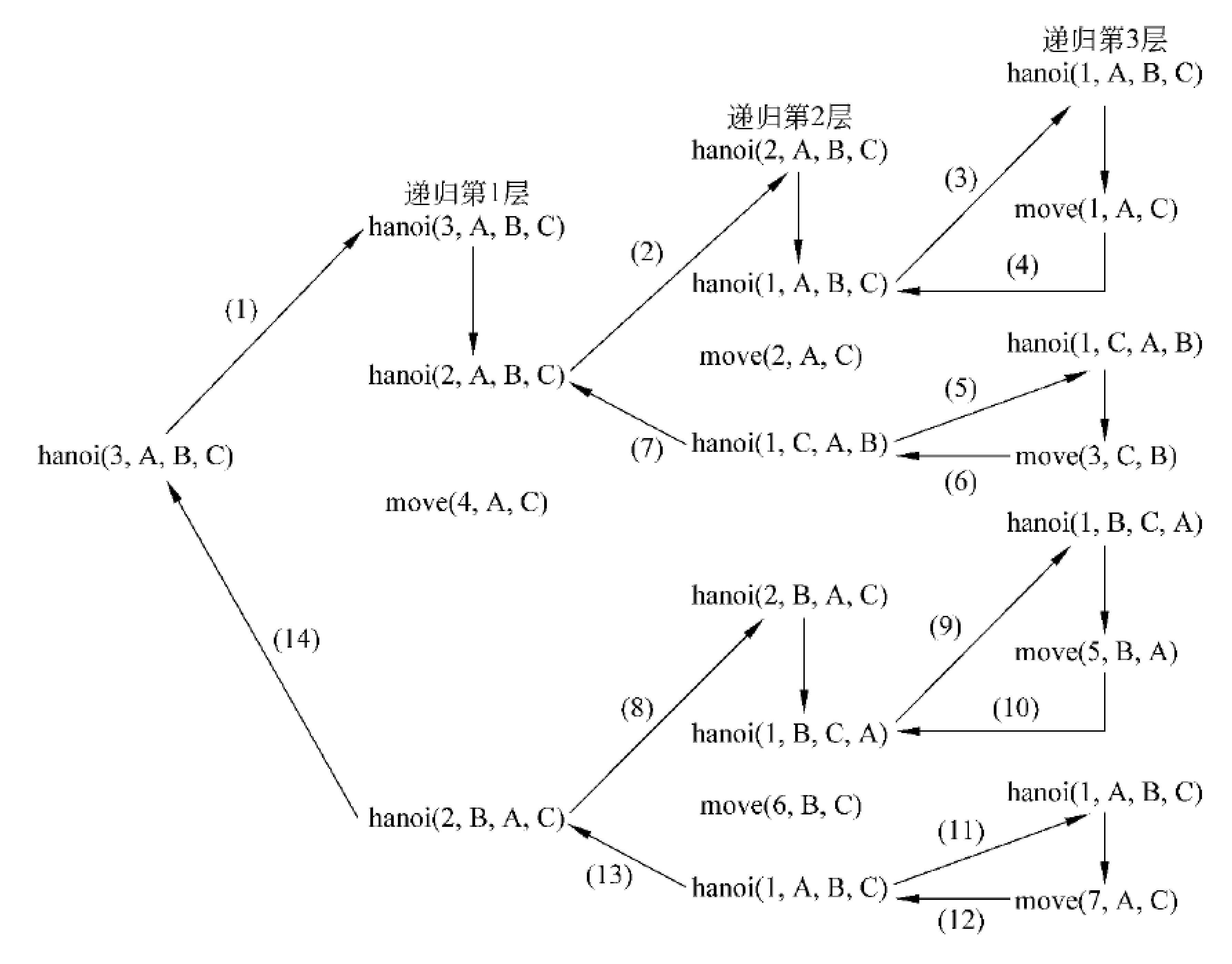


图 8.5　n＝3 时函数的递归调用过程

8.5　变量的作用域

当程序中有多个函数时，定义的每个变量只能在一定的范围内访问，称之为变量的作用域。按作用域划分，可以将变量分为局部变量和全局变量。

8.5.1 局部变量

在一个函数内或者语句块内定义的变量称为局部变量。局部变量的作用域仅限于定义它的函数体或语句块中，任意一个函数都不能访问其他函数中定义的局部变量。因此，在不同的函数之间可以定义同名的局部变量，虽然同名但却代表不同的变量，但不会发生命名冲突。例如：

```
def fun1(a):
    x=a+10
    …
def fun2(a,b):
    x,y=a,b
    …
```

说明：

(1) fun1()函数中定义了形参 a 和局部变量 x，fun2()函数中定义了形参 a、b 和局部变量 x、y，这些变量各自在定义它们的函数体中有效，其作用范围都限定在各自的函数中。

(2) 在不同的函数中定义的变量，即使使用相同的变量名也不会互相干扰、互相影响。例如，fun1()函数和 fun2()函数都定义了变量 a 和 x，变量名相同，但作用范围不同。

(3) 形参也是局部变量，例如，fun1 函数中的形参 a。

【例 8.10】 局部变量应用。

程序如下：

```
def fun(x):
    print("x=",x)
    x=20
    print("changed local x=",x)


x=30
fun(30)
print("main x=",x)
```

程序运行结果：

```
x=30
changed local x=20
main x=30
```

在 fun()函数中，第一次输出 x 的值，x 是形参，值由实参传递而来，是 30，接着执行赋值语句 x=20 后，再次输出 x 的值是 20。主调函数中，x 的值为 30，当调用 fun()函数时，该值不受影响，因此主调函数输出 x 的值是 30。

8.5.2 全局变量

在所有函数之外定义的变量称为全局变量,它可以在多个函数中被引用。例如:

```
m=1                          #定义为全局变量
def fun1(a):
    print(m)                 #使用全局变量
    …
n=1                          #定义 n为全局变量
def fun2(a,b):
    n=a* b                   #使用局部变量
    …
```

变量 m 和 n 为全局变量,在函数 fun1()和函数 fun2()中可以直接引用。

【例 8.11】 全局变量应用。

程序如下:

```
x = 30
def func():
    global x
    print('x的值是',x)
    x = 20
    print('全局变量 x改为',x)
func()
print('x的值是',x)
```

程序运行结果:

```
x的值是 30
全局变量 x改为 20
x的值是 20
```

8.6 模　　块

随着程序规模越来越大,如何将一个大型文件分割成几个小文件就变得很重要。为了使代码看起来更优美、紧凑、容易修改、适合团体开发,Python 引入了模块的概念。

在 Python 中,可以把程序分割成许多单个文件,这些单个的文件就称为模块。模块就是将一些常用的功能单独放置到一个文件中,方便其他文件来调用。前面编写代码时保存的以.py 为扩展名的文件,都是独立的模块。

8.6.1 定义模块

与函数类似,从用户的角度看,模块也分为标准库模块和用户自定义模块。

1. 标准库模块

标准库模块是 Python 自带的函数模块。Python 提供了大量的标准库模块，实现了很多常用的功能。Python 标准库提供了文本处理、文件处理、操作系统功能、网络通信、网络协议等功能。

(1) 文本处理：包含文本格式化、正则表达式匹配、文本差异计算与合并、Unicode 支持和二进制数据处理等功能。

(2) 文件处理：包含文件基本操作、创建临时文件、文件压缩与归档、操作配置文件等功能。

(3) 操作系统功能：包含线程与进程支持、I/O 复用、日期与时间处理、调用系统函数、日志等功能。

(4) 网络通信：包含网络套接字、SSL 加密通信、异步网络通信等功能。

(5) 网络协议：支持 HTTP、FTP、SMTP、POP、IMAP、NNTP、XMLRPC 等多种网络协议，并提供了编写网络服务器的框架。

(6) 其他功能，包括国际化支持、数学运算、Hash、Tkinter 等。

另外，Python 还提供了大量的第三方模块，使用方式与标准库类似。它们的功能覆盖科学计算、Web 开发、数据库接口、图形系统多个领域。

2. 用户自定义模块

用户建立一个模块就是建立扩展名为.py 的 Python 程序。例如，在一个 module.py 的文件中输入 def 语句，就生成了一个包含属性的模块。

```
def printer(x)
    print(x)
```

当模块导入的时候，Python 把内部的模块名映射为外部的文件名。

8.6.2 导入模块

导入模块就是给出一个访问模块提供的函数、对象和类的方法。模块的导入有如下三种。

1. 导入模块的方法

导入模块的一般形式为：

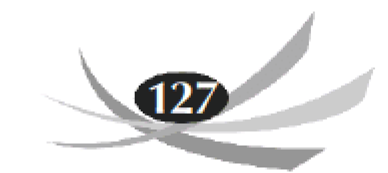
```
import 模块
```

用 import 语句直接导入模块，就在当前的名字空间(namespace)建立了一个到该模块的引用。这种引用必须使用全称，当使用在被导入模块中定义的函数时，必须包含模块的名字。

【例 8.12】 求列表中值为偶数的和。

程序如下：

```
def func_sum(a_list):
```




```
    s=0
    for i in range(0,len(a_list)):
            if a_list[i]%2==0:
              s=s+a_list[i]
    return s
```

再写一个文件导入上面的模块：

```
#exp8.12.py
import evensum
a_list=[3,54,65,76,45,34,100,-2]
s=evensum.func_sum(a_list)
print("sum=",s)
```

程序运行结果：

```
sum=262
```

2. 导入模块中的函数

导入模块中的函数的一般形式为：

```
from模块名 import 函数名
```

通过这种方法，函数名被直接导入到本地名字空间中去，所以它可以直接使用，而不需要加上模块名的限定表示。

例 8.12 中导入模块的文件可改写为：

```
from evensum import func_sum
a_list=[3,54,65,76,45,34,100,-2]
s=func_sum(a_list)
print("sum=",s)
```

3. 导入模块中的所有函数

导入模块中的所有函数的一般形式为：

```
from模块名 import *
```

同第二种导入方法，它只是一次导入模块中的所有函数，不需要一一列举函数名。

8.7　函数应用举例

【例 8.13】 采取插入排序法将 10 个数据按从小到大的顺序进行排序。

分析：插入排序的基本操作是每一步都将一个待排数据按其大小插入到已经排序的数据中的适当位置，直到全部插入完毕。插入算法把要排序的数组分成两部分：第一部分包含了这个数列表中，除最后一个元素外的所有元素，而第二部分就只包含这一个元素(即待插入元素)。在第一部分排序完成后，再将这个最后元素插入到已排好序的第一部

分中。排序过程如下。

(1) 假设当前需要排序的元素(array[i]),跟已经排序好的最后一个元素比较(array[i−1]),如果满足条件继续执行后面的程序,否则循环到下一个要排序的元素。

(2) 缓存当前要排序的元素的值,以便找到正确的位置进行插入。

(3) 排序的元素跟已经排好序的元素比较,比它大的向后移动。

(4) 把要排序的元素,插入到正确的位置。

程序如下:

```
def insert_sort(array):
  for i in range(1,len(array)):
    if array[i-1] > array[i]:
        temp = array[i]            #当前需要排序的元素暂存到 temp 中
        index = i                  #用来记录排序元素需要插入的位置
        while index > 0 and array[index-1] > temp:
            array[index] = array[index-1]
                                   #把已经排好序的元素后移一位,留下需要插入的位置
            index-=1
        array[index] = temp        #把需要排序的元素插入到指定位置

b=input("请输入一组数据:")
array=[]
for i in b.split(','):
    array.append(int(i))
print("排序前的数据:")
print(array)
insert_sort(array)                 #调用 insert_sort()函数
print("排序后的数据:")
print(array)
```

程序运行结果:

```
请输入一组数据:100,43,65,101,54,65,4,2017,123,55
排序前的数据:
[100,43,65,101,54,65,4,2017,123,55]
排序后的数据:
[4,43,54,55,65,65,100,101,123,2017]
```

【例 8.14】 用递归的方法求 x^n。

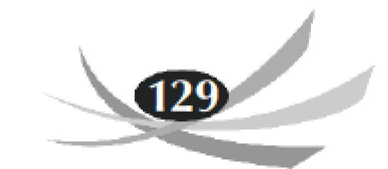
分析:求 x^n 可以使用下面的公式。

$$x^n = \begin{cases} 1 & n = 0 \\ n * x^{n-1} & n > 0 \end{cases}$$

递推归纳:$x^n \to x^{n-1} \to x^{n-2} \to \cdots \to x^2 \to x^1$。

递归终止条件:当 $n=0$ 时,$x^0=1$。

程序如下:




```
def xn(x,n):
    if n==0:
        f=1
    else:
        f=x*xn(x,n-1)
    return f

x,n=eval(input("please input x and n"))
if n<0:
    n=-n
    y=xn(x,n)
    y=1/y
else:
    y=xn(x,n)
print(y)
```

程序运行结果：

```
please input x and n: 3,5
243
```

再次运行程序，结果如下：

```
please input x and n: 3,-5
0.004115226337448556
```

【例 8.15】 计算从公元 1 年 1 月 1 日到 y 年 m 月 d 日的天数(含两端)。例如，从公元 1 年 1 月 1 日到 1 年 2 月 2 日的天数是 31+2=33。

分析：要计算从公元 1 年 1 月 1 日到 y 年 m 月 d 日的天数，可以分为三步完成。

(1) 计算从公元 1 年到 y−1 年这些整年的天数，每年是 365 天或 366 天(闰年是 366 天)。

(2) 对于第 y 年，当 m>1 时，先计算 1～m−1 月整月的天数。

(3) 最后加上零头(第 m 月的 d 天)即可。

程序如下：

```
#判断某年是否为闰年
def  leapYear(y):
    if y<1:
        y=1
    if ((y % 400)==0 or (y % 4)==0 and (y % 100) !=0):
        lp=1
    else:
        lp=0
    return lp

#计算 y年 m月的天数
```

```
def  getLastDay(y,m):
    if y<1:
        y=1
    if m<1:
        m=1
    if m>12:
        m=12
    #每个月的正常天数
    #月份    1  2  3  4  5  6  7  8  9 10  11  12
    monthDay=[31,28,31,30,31,30,31,31,30,31,30,31]
    r=monthDay[m-1]
    if m==2:
        r =r+leapYear(y)      #处理闰年的 2月天数
    return r

#计算从公元 1年 1月 1日到 y年 m月 d日的天数(含两端的天数)
def calcDays(y,m,d):
    if y<1:
        y=1
    if m<1:
        m=1
    if m>12:
        m=12
    if d<1:
        d=1
    if d>getLastDay(y,m):
        d=getLastDay(y,m)
    T=0
    for i in range(1,y):
        T =T+365+leapYear(i)
    for i in range(1,m):
        T =T+getLastDay(y,i)
    T =T+d
    return T

y,m,d =eval(input("input year,month,day:"))
days =calcDays(y,m,d)
print("从 1年 1月 1日到",y,"年",m,"月",d,"日 共",days,"天")
```

程序运行结果：

```
input year,month,day:2017,1,1
从 1年 1月 1日到 2017 年 1月 1日 共 736330 天
```

习　　题

1. 编写一个程序，已知一个圆筒的半径、外径和高，计算该圆筒的体积。

2. 编写一个求水仙花数的函数，求 100～999 的全部水仙花数。

3. 编写一个函数，输出整数 m 的全部素数因子。例如，m＝120，等数因子为 2，2，2，3，5。

4. 编写一个函数，求 10 000 以内所有的完数。所谓完数是指一个数正好是它的所有约数之和。例如，6 就是一个完数，因为 6 的因子有 1、2、3，并且 6＝1＋2＋3。

5. 如果有两个数，每一个数的所有约数（除它本身以外）的和正好等于对方，则称这两个数为互满数。求出 10 000 以内所有的互满数，并显示输出，并使用函数实现求一个数和它的所有约数（除它本身）的和。

6. 用递归函数求 $s=\sum_{i=1}^{n} i$ 的值。

第 9 章 文　件

在前面的章节中我们使用的原始数据很多都是通过键盘输入的，并将输入的数据放入指定的变量中，若要处理（运算、修改、删除、排序等）这些数据，可以从指定的变量中取出并进行处理。但在数据量大、数据访问频繁以及数据处理结果需要反复查看或使用时，就有必要将程序的运行结果保存下来。为了解决以上问题，在 Python 中引入了文件，将这些待处理的数据存储在指定的文件中，当需要处理文件中的数据时，可以通过文件处理函数，取得文件内的数据并存放到指定的变量中进行处理，数据处理完毕后再将数据存回指定的文件中。有了对文件的处理，不但数据容易维护，而且同一个程序可处理数据格式相同但文件名不同的文件，增加了程序的使用弹性。

文件操作是一种基本的输入与输出方式，数据以文件的形式进行存储，操作系统以文件为单位对数据进行管理。本章主要介绍文件的基本概念、文件的操作方法及文件操作的应用。

9.1　文件概述

9.1.1　文件的基本概念

文件是指存放在外部存储介质（可以是磁盘、光盘、磁带等）上的一组相关信息的集合。操作系统以文件形式管理外部存储介质上的数据。当打开一个文件或者创建一个新文件时，一个数据流和一个外部文件（也可能是一个物理设备）相关联。为标识一个文件，每个文件都必须有一个文件名作为访问文件的标志，其一般结构为：

主文件名[.扩展名]

通常情况下，文件应该包括盘符名、路径、主文件名和扩展名四部分信息。实际上，在前面的各章中已经多次使用了文件，例如源程序文件、库文件（头文件）等。程序在内部存储介质（简称内存）中运行的过程中与外部存储介质（简称外存）交互主要通过以下两种方法进行。

（1）以文件为单位将数据写到外存中。

（2）从外存中根据文件名读取文件中的数据。

也就是说，要想读取外存中的数据，必须先按照文件名找到相应的文件，然后再从文件中读取数据；要想将数据存放到外存中，首先要在外存上建立一个文件，然后再向该文件中写入数据。

可以从不同的角度对文件进行分类，分别如下所述。

（1）根据文件依附的介质，可分为普通文件和设备文件。

普通文件是指驻留在磁盘或其他外存上的一个有序数据集，可以是源文件、目标文件、可执行程序，也可以是一组待输入处理的原始数据，或者是一组输出的结果。对于源文件、目标文件、可执行程序可以称作程序文件，对输入和输出数据则可称作数据文件。

设备文件是指与主机相连的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看作是一个文件来进行管理，把它们的输入和输出等同于对磁盘文件的读和写。

（2）根据文件的组织形式，可分为顺序读写文件和随机读写文件。

顾名思义，顺序读写文件是指按从头到尾的顺序读出或写入的文件。通常在重写整个文件操作时，使用顺序读写；而要更新文件中某个数据时，不使用顺序读写。此种文件每次读写的数据长度不等，较节省空间，但查询数据时都必须从第一个记录开始找，较费时间。

随机读写文件大都使用结构方式来存放数据，即每个记录的长度是相同的，因而通过计算便可以直接访问文件中的特定记录，也可以在不破坏其他数据的情况下把数据插入到文件中，是一种跳跃式直接访问方式。

（3）根据文件的存储形式，可分为文本文件和二进制文件。

文本文件也称 ASCII 文件，这种文件在磁盘中存放时每个字符对应一个字节，用于存放对应的 ASCII 码。

例如，数 1124 的存储形式如下。

```
ASCII 码：   00110001  00110001  00110010  00110100
                ↓         ↓         ↓         ↓
十进制码：      1         1         2         4
```

共占用 4 个字节。ASCII 文件可在屏幕上按字符显示，例如，源程序文件就是 ASCII 文件，用 DOS 命令中的 TYPE 可显示文件的内容。由于是按字符显示，因此能读懂文件内容。

二进制文件是按二进制的编码方式来存放文件的。例如，数 1124 的存储形式为：

```
00000100  01100100
```

只占两个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。Python 在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。

ASCII 文件和二进制文件的主要区别在于：

（1）从存储形式上看，二进制文件是按该数据类型在内存中的存储形式存储的，而 ASCII 文件则将该数据类型转换为可在屏幕上显示的形式存储的。

（2）从存储空间上看，ASCII 文件存储方式所占的空间比较多，而且所占的空间大小与数值大小有关。

（3）从读写时间上看，由于 ASCII 文件在外存上是以 ASCII 码存放，而在内存中的数据都是以二进制存放的，所以，当进行文件读写时，要进行转换，造成存取速度较慢。对于二进制文件来说，数据就是按其在内存中的存储形式在外存上存放的，所以不需要进行

这样的转换，在存取速度上较快。

(4) 从作用上看，由于 ASCII 文件可以通过编辑程序，如 edit、记事本等，进行建立和修改，也可以通过 DOS 中的 TYPE 命令显示出来，因而 ASCII 文件通常用于存放输入数据及程序的最终结果。而二进制文件则不能显示出来，所以用于暂存程序的中间结果，供另一段程序读取。

在 C 语言中，标准输入设备(键盘)和标准输出设备(显示器)是作为 ASCII 文件处理的，它们分别称为标准输入文件和标准输出文件。

9.1.2 文件的操作流程

文件的操作包括对文件本身的基本操作和对文件中信息的处理。首先，只有通过文件指针，才能调用相应的文件，然后才能对文件中的信息进行操作，进而达到从文件中读数据或向文件中写数据的目的。具体涉及的操作有：建立文件、打开文件、从文件中读数据或向文件中写数据、关闭文件等。一般的操作步骤如下。

(1) 建立/打开文件。

(2) 从文件中读取数据或者给文件中写数据。

(3) 关闭文件。

打开文件是进行文件的读或写操作之前的必要步骤。打开文件就是将指定文件与程序联系起来，为下面进行的文件读写工作做好准备。如果不打开文件就无法读写文件中的数据。当为进行写操作而打开一个文件时，如果这个文件存在，则打开它；如果这个文件不存在，则系统会新建这个文件，并打开它。当为进行读操作而打开一个文件时，如果这个文件存在，则系统打开它；如果这个文件不存在，则出错。数据文件可以借助常用的文本编辑程序建立，就如同建立源程序文件一样，当然，也可以是其他程序写操作生成的文件。

从文件中读取数据，就是从指定文件中取出数据，存入程序在内存中的数据区，如变量或序列中。

向文件中写数据，就是将程序中的数据存储到指定的文件中，即文件名所指定的存储区中。

关闭文件就是取消程序与指定的数据文件之间的联系，表示文件操作的结束。

9.2 文件的打开与关闭

9.2.1 打开文件

在对文件进行读写操作之前要先打开文件。所谓打开文件，实际上是建立文件的各种有关信息，并使文件指针指向该文件，以便进行其他操作。

1. open()函数

Python 中使用 open()函数来打开文件并返回文件对象，其一般调用格式为：

```
文件对象=open(文件名[,打开方式][,缓冲区])
```

函数参数说明如下。

open()函数的第一个参数是传入的文件名，可以包含盘符、路径和文件名。如果只有文件名，没有带路径的话，那么 Python 会在当前文件夹中去找到该文件并打开。

第二个参数“打开方式”是可选参数，表示打开文件后的操作方式，文件打开方式使用具有特定含义的符号表示，如表 9.1 所示。

表 9.1　文件的打开方式

文件使用方式	含　义
rt	只读打开一个文本文件，只允许读数据
wt	只写打开或建立一个文本文件，只允许写数据
at	追加打开一个文本文件，并在文件末尾写数据
rb	只读打开一个二进制文件，只允许读数据
wb	只写打开或建立一个二进制文件，只允许写数据
ab	追加打开一个二进制文件，并在文件末尾写数据
rt+	读写打开一个文本文件，允许读和写
wt+	读写打开或建立一个文本文件，允许读和写
at+	读写打开一个文本文件，允许读，或在文件末尾追加数据
rb+	读写打开一个二进制文件，允许读和写
wb+	读写打开或建立一个二进制文件，允许读和写
ab+	读写打开一个二进制文件，允许读，或在文件末尾追加数据

第三个参数“缓冲区”也是可选参数，表示文件操作是否使用缓冲存储方式，取值有 0，1，−1 和大于 1 四种。如果缓冲区参数被设置为 0，则表示缓冲区关闭（只适用于二进制模式），不使用缓冲区；如果缓冲区参数被设置为 1，则表示使用缓冲存储（只适用于文本模式）；如果缓冲区参数被设置为−1，则表示使用缓冲存储，并且使用系统默认缓冲区的大小；如果缓冲区参数被设置为大于 1 的整数，则表示使用缓冲存储，并且该参数指定了缓冲区的大小。

假设有一个名为 somefile.txt 的文本文件，存放在 c:\text 下，那么可以这样打开文件：

```
>>> x = open('c:\\text\\somefile.txt','r',buffering=1024)
```

注意：文件打开成功，没有任何提示。

对于文件打开方式有以下几点说明。

(1) 文件打开方式由 r、w、a、t、b、+六个字符拼成，各字符的含义是：

r(read)：读。

w(write)：写。

a(append)：追加。

t(text)：文本文件，可省略不写。

b(binary)：二进制文件。

+：读和写。

(2) 用 r 方式打开一个文件时，该文件必须已经存在，且只能从该文件读取。

(3) 用 w 方式打开的文件只能向该文件写入。若打开的文件不存在，则以指定的文件名建立该文件；若打开的文件已经存在，则将该文件删去，重建一个新文件。

(4) 若要向一个已存在的文件中追加新的信息，只能用 a 方式打开文件。若此时该文件不存在，则会新建一个文件。

使用 open()函数成功打开一个文件之后，会返回一个文件对象，得到这个文件对象，就可以读取或修改该文件了。

2. 文件对象属性

文件一旦被打开，就可以通过文件对象的属性得到该文件的有关信息。常用的文件对象属性如表 9.2 所示。

表 9.2　常用的文件对象属性

文件对象属性	含　义
name	返回文件的名称
mode	返回文件的打开方式
closed	如果文件被关闭返回 True，否则返回 False

文件属性的引用方法为：

```
文件对象名.属性名
```

例如：

```
>>> fp=open("e:\\qq.txt","r")
>>> fp.name
'e:\\qq.txt'
>>> fp.mode
'r'
>>> fp.closed
False
```

3. 文件对象方法

打开文件并取得文件对象之后，就可以利用文件对象方法对文件进行读取或修改等操作。表 9.3 列举了常用的文件对象方法。

表 9.3　常用的文件对象方法

文件对象方法	含　　义
close()	关闭文件，并将属性 closed 设置为 True
read(count)	从文件对象中读取至多 count 个字节，如果没有指定 count，则读取从当前文件指针直至文件末尾
readline(count)	从文件中读取一行内容
readlines(sizehint)	读取文件的所有行(直到结束符 EOF)，也就是整个文件的内容，把文件每一行作为列表的成员，并返回这个列表
write(string)	将字符串 string 写入到文件
writelines(seq)	将字符串序列 seq 写入到文件，seq 是一个返回字符串的可迭代对象
seek(offset,whence)	把文件指针移动到相对于 whence 的 offset 位置。whence 为 0 表示文件开始处；为 1 表示当前位置；为 2 表示文件末尾
next()	返回文件的下一行，并将文件操作标记移到下一行
tell()	返回当前文件指针位置(相对文件起始处)
flush()	清空文件对象，并将缓存中的内容写入磁盘(如果有)

9.2.2　关闭文件

当一个文件使用结束时，就应该关闭它，以防止其被误操作而造成文件信息的破坏和文件信息的丢失。关闭文件是断开文件对象与文件之间的关联，此后不能再继续通过该文件对象对该文件进行读写操作。Python 使用 close()函数关闭文件。close()函数的一般形式为：

```
文件对象名.close()
```

9.3　文件的读写

9.3.1　文本文件的读写

1. 文本文件的读取

Python 对文本文件的读取是通过调用文件对象方法来实现的。文件对象提供了三种读取方法：read()、readline()和 readlines()。

1）read()方法

read()方法的一般形式为：

```
文件对象.read()
```

其功能是读取从当前位置直到文件末尾的内容，该方法通常将读取的文件内容存放到一个字符串变量中。

假如有一个文本文件 file1.txt，其内容如下：

```
There were bears everywhere.
They were going to Switzerland.
```

采用 read()方法读该文件内容,结果如下:

```
fp=open("e:\\file1.txt","r")            #以只读的方式打开 file1.txt 文件
string1=fp.read()
>>>print("Read Line: %s" % (string1))
Read Line: There were bears everywhere.
They were going to Switzerland.
```

read()方法也可以带有参数,一般形式为:

```
文件对象.read([size])
```

其功能是从文件当前位置起读取 size 个字节,返回结果是一个字符串。如果 size 大于文件从当前位置开始到末尾的字节数,则读取到文件结束为止。例如:

```
fp=open("e:\\file1.txt","r")            #以只读的方式打开 file1.txt 文件
string2=fp.read(10)                     #读取 10 个字节
>>>print("Read Line: %s" % (string2))
Read Line: There were
```

2) readline()方法

readline()方法的一般形式为:

```
文件对象.readline()
```

其功能是读取从当前位置到行末的所有字符,包括行结束符,即每次读取一行,当前位置移到下一行。如果当前处于文件末尾,则返回空串。例如:

```
>>> fp=open("e:\\file1.txt","r")
>>> string3=fp.readline()
>>>print("Read Line: %s" % (string3))
Read Line: There were bears everywhere.
```

3) readlines()方法

readlines()方法的一般形式为:

```
文件对象.readlines()
```

其功能是读取从当前位置到文件末尾的所有行,并将这些行保存在一个列表(list)变量中,每行作为一个元素。如果当前文件处于文件末尾,则返回空列表。例如:

```
>>> fp=open("e:\\file1.txt","r")
>>> string4=fp.readlines()
>>>print("Read Line: %s" % (string4))
Read Line: ['There were bears everywhere.\n','They were going to Switzerland.']
```

```
>>> string5= fp.readlines()            #再次读取文件,返回空列表
>>> print("Read Line: %s" % (string5))
Read Line: []
```

2. 文本文件的写入

文本文件的写入通常使用 write()方法,有时也使用 writelines()方法。

1) write()方法

write()方法的一般形式为:

```
文件对象.write(字符串)
```

其功能是在文件当前位置写入字符串,并返回写入的字符个数。例如:

```
>>> fp.open("e:\\file1.txt","w")       #以写方式打开 file1.txt 文件
>>> fp.write("Python")                 #将字符串"Python"写入文件 file1.txt 文件
6
>>> fp.write("Python programming")
18
>>> fp.close()
```

2) writelines()方法

writelines()方法的一般形式为:

```
文件对象.writelines(字符串元素的列表)
```

其功能是在文件的当前位置处依次写入列表中的所有元素。例如:

```
>>> fp.open("e:\\file1.txt","w")
>>> fp.writelines(["Python","Python programming"])
```

【例 9.1】 把一个包含两列内容的文件 input.txt,分割成两个文件 col1.txt 和 col2.txt,每个文件有一列内容。input.txt 文件的内容如图 9.1 所示。

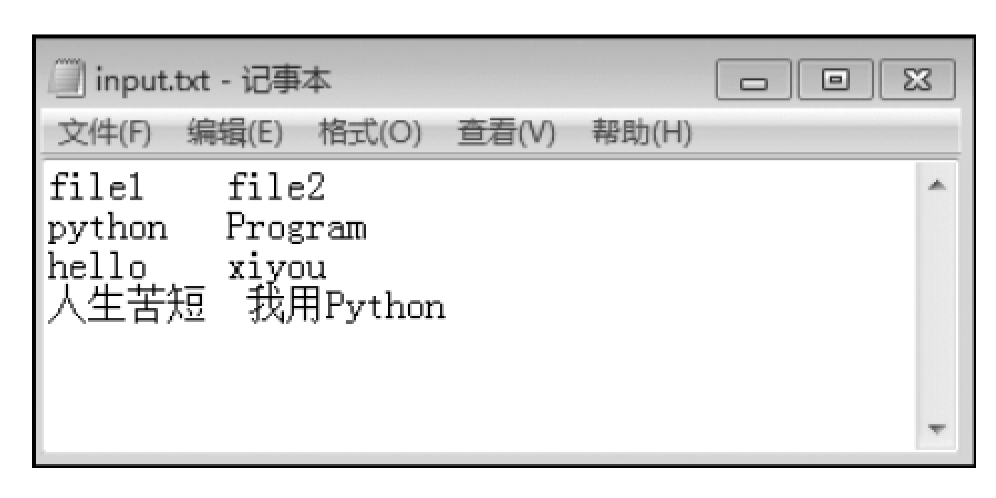


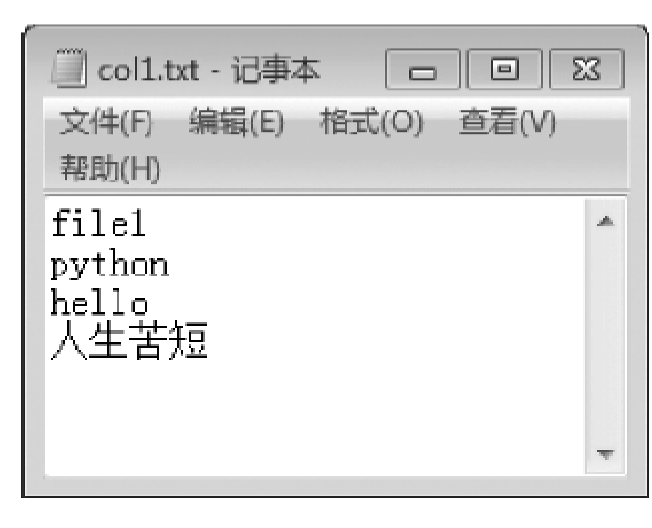
图 9.1　input.txt 文件的内容

程序如下:

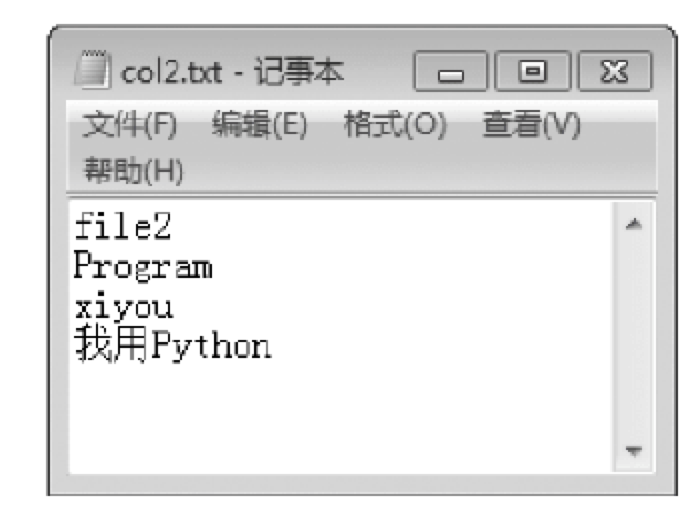
```
def split_file(filename):              #把文件分成两列
    col1 = []
    col2 = []
```

```
    fd = open(filename)                 #打开文件
    text = fd.read()                    #读入文件的内容
    lines = text.splitlines()           #把读入的内容分行
    for line in lines:
        part = line.split(None,1)
        col1.append(part[0])
        col2.append(part[1])
    return col1,col2
def write_list(filename,alist):         #把文字列表内容写入文件
    fd= open(filename,'w')
    for line in alist:
        fd.write(line+ '\n')
filename= 'input.txt'
col1,col2= split_file(filename)
write_list('col1.txt',col1)
write_list('col2.txt',col2)
```

程序运行结果如图 9.2 所示。



(a) col1.txt文件内容



(b) col2.txt文件内容

图 9.2　例 9.1 运行结果

9.3.2　二进制文件的读写

前面介绍的读写方法,读写的都是字符串,对于其他类型数据则需要转换。Python 中 struct 模块中的 pack()和 unpack()方法可以进行转换。

1. 二进制文件的写入

Python 中二进制文件的写入有两种方法:一种是通过 struct 模块的 pack()方法把数字和布尔值转换成字符串,然后用 write()方法写入二进制文件中;另一种是用 pickle 模块的 dump()方法直接把对象转换为字符串并存入文件中。

1) pack()方法

pack()方法的一般形式为:

```
pack(格式串,数据对象表)
```

其功能是将数字转换为二进制的字符串。格式串中的格式字符如表 9.4 所示。

表 9.4　格式字符

格式字符	C 语言类型	Python 类型	字节数
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer or long	4
l	long	integer	4
L	unsigned long	long	4
q	long long	long	8
Q	unsigned long long	long	8
f	float	float	4
d	double	float	8
s	char[]	string	1
p	char[]	string	1
P	void *	long	与操作系统的位数有关

pack()方法使用如下：

```
>>> import struct
>>> x=100
>>> y=struct.pack('i',x)          #将 x 转换成二进制字符串
>>> y                             #输出转换后的字符串 y
b'd\x00\x00\x00'
>>> len(y)                        #计算 y 的长度
4
```

此时，y 是一个 4 字节的字符串。如果要将 y 写入文件，可以这样实现：

```
>>> fp=open("e:\\file2.txt","wb")
>>> fp.write(y)
4
>>> fp.close()
```

【例 9.2】 将一个整数、一个浮点数和一个布尔型对象存入一个二进制文件中。

分析：整数、浮点数和布尔型对象都不能直接写入二进制文件，需要使用 pack()方法

将它们转换成字符串再写入二进制文件中。

程序如下：

```
import struct
i=12345
f=2017.2017
b=False
string=struct.pack('if?',i,f,b)
                                    #将整数 i、浮点数 f 和布尔对象 b 依次转换为字符串
fp=open("e:\\string1.txt","wb")     #打开文件
fp.write(string)                    #将字符串 string 写入文件
fp.close()                          #关闭文件
```

运行时在 e 盘下创建 string1. txt 文件，运行结束后打开 string. txt 文件，其内容如图 9. 3 所示。

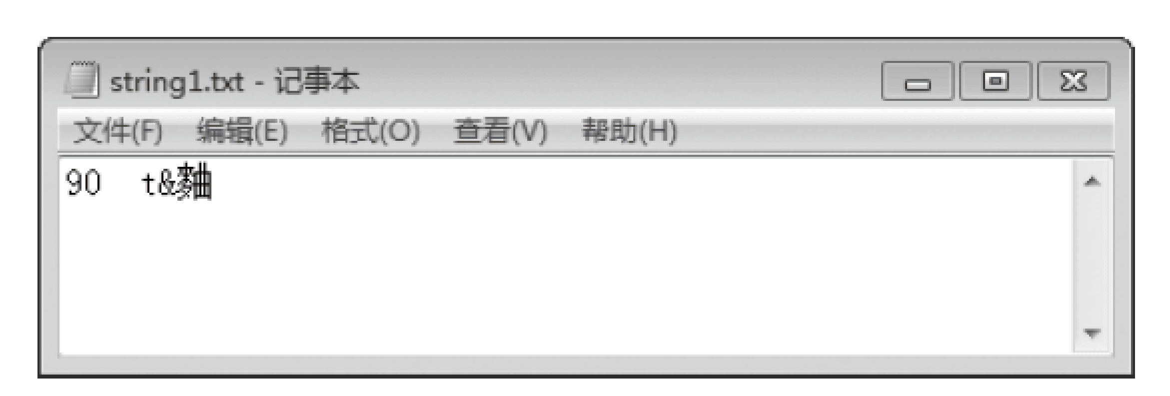


图 9. 3 string1. txt 文件内容

2） dump()方法

dump ()方法的一般形式为：

```
dump(数据,文件对象)
```

其功能是将数据对象转换成字符串，然后再保存到文件中。其用法如下：

```
>>> import pickle
>>> x=100
>>> fp=open("e:\\file3.txt","wb")
>>> pickle.dump(x,fp)               #把整数 x 转换成字符串并写入文件中
>>> fp.close()
```

【例 9. 3】 用 dump()方法实现例 9. 2。

程序如下：

```
import pickle
i=12345
f=2017.2017
b=False
fp=open("e:\\string2.txt","wb")
pickle.dump(i,fp)
pickle.dump(f,fp)
pickle.dump(b,fp)
```

```
fp.close()
```

2. 二进制文件的读取

读取二进制文件的内容应根据写入时的方法而采取相应的方法进行。使用 pack()方法写入文件的内容应该使用 read()方法读出相应的字符串，然后通过 unpack()方法还原数据；使用 dump()方法写入文件的内容应使用 pickle 模块的 load()方法还原数据。

1）unpack()方法

unpack()方法的一般形式是：

```
unpack(格式串,字符串表)
```

其功能与 pack()正好相反，将“字符串表”转换成“格式串”（如表 9.1 所示）指定的数据类型。该方法返回一个元组。例如：

```
>>> import struct
>>> fp=open("e:\\file2.txt","rb")        #以只读方式打开 file2.txt 文件
>>> y=fp.read()
>>> x=struct.unpack('i',y)
>>> x
(100,)
```

【例 9.4】 读取例 9.2 中写入的 string1.txt 文件内容。

分析：string1.txt 中存放的是字符串，需要先使用 read()方法读取每个数据的字符串形式，然后进行还原。

程序如下：

```
import struct
fp=open("e:\\string1.txt","rb")
string=fp.read()
a_tuple=struct.unpack('if?',string)
print("a_tuple=",a_tuple)
i=a_tuple[0]
f=a_tuple[1]
b=a_tuple[2]
print("i=%d,f=%f"%(i,f))
print("b=",b)
fp.close()
```

程序运行结果：

```
a_tuple=(12345,2017.20166015625,False)
i=12345,f=2017.201660
b=False
```



2）load()方法

load()方法的一般形式为：

```
load(文件对象)
```

其功能是从二进制文件中读取字符串，并将字符串转换为 Python 的数据对象。该方法返回还原后的字符串。例如：

```
>>> import pickle
>>> fp=open("e:\\file3.txt","rb")
>>> x=pickle.load(fp)
>>> fp.close()
>>> x                                    #输出读出的数据
100
```

【例 9.5】 读取例 9.3 中写入的 string2.txt 文件内容。

分析：在例 9.3 中，向 string2.txt 文件中写入了一个整型、一个浮点型、一个布尔型数据，每次读取之后需要判断读到文件末尾。

程序如下：

```
import pickle
fp=open("e:\\string2.txt","rb")
while True:
  n=pickle.load(fp)
  if(fp):
      print(n)
  else:
      break
fp.close()
```

程序运行结果：

```
12345
2017.2017
True
```

9.4 文件的定位

在实际问题中常要求只读写文件中某一段指定的内容，为了解决这个问题，可以移动文件内部的位置指针到需要读写的位置，再进行读写，这种读写称为随机读写。实现文件的随机读写关键是要按要求移动位置指针，这个过程称为文件的定位。Python 中文件的定位提供了以下几种方法。

1. tell()方法

tell()方法的一般形式为：

```
文件对象.tell()
```

其功能是获取文件的当前指针位置，即相对于文件开始位置的字节数。例如：

```
>>> fp=open("e:\\file1.txt","r")
>>> fp.tell()                    #文件打开之后指针位于文件的开始处,即位于第一个字符
0
>>> fp.read(10)                  #从当前位置起读取10个字节内容
>>> fp.tell()                    #返回读取10个字节内容之后的文件位置
10
```

2. seek()方法

seek()方法的一般形式为：

```
文件对象.seek(offset,whence)
```

其功能把文件指针移动到相对于 whence 的 offset 位置。其中，offset 表示要移动的字节数，移动时以 offset 为基准，offset 为正数表示向文件末尾方向移动，为负数表示向文件开头方向移动；whence 指定移动的基准位置，如果设置为 0 表示以文件开始处作为基准点，设置为 1 表示以当前位置为基准点，设置为 2 表示以文件末尾作为基准点。例如：

```
>>> fp=open("e:\\file1.txt","rb")   #以二进制方式打开文件
>>> fp.read()                       #读取整个文件内容,文件指针移动到文件末尾
b'PythonPython programming'
>>> fp.read()                       #再次读取文件内容,返回空串
b''
>>> fp.seek(0,0)                    #以文件开始作为基准点,向文件末尾方向移动0个字节
0
>>> fp.read()                       #文件指针移动之后再次读取
b'PythonPython programming'
>>> fp.seek(6,0)                    #以文件开始作为基准点,向文件末尾方向移动6个字节
6
>>> fp.read()                       #文件指针移动之后再次读取
b'Python programming'
>>> fp.seek(-11,2)                  #以文件末尾作为基准点,向文件开头方向移动11个字节
13
>>> fp.read()                       #文件指针移动之后再次读取
b'programming'
```

【例 9.6】 编写程序，求取文件指针位置及文件长度。

程序如下：

```
filename=input("请输入文件名:")
fp=open(filename,"r")              #以只读方式打开文件
curpos=fp.tell()                   #获取文件当前指针位置
print("the begin of %s is %d"%(filename,curpos))
fp.seek(0,2)        #以文件末尾作为基准点,向文件头方向移动 0个字节,即文件指针移动到
                    #文件尾部
length=fp.tell()
print("the end begin of %s is %d"%(filename,length))
```

9.5 与文件相关的模块

Python 模块(module)是一个 Python 文件,以.py 为扩展名,包含了 Python 对象定义和 Python 语句。模块可以定义函数、类和变量,模块里也可以包含可执行的代码。Python 中对文件、目录的操作需要用到 os 模块和 os.path 模块。

9.5.1 os 模块

Python 内置的 os 模块提供了访问操作系统服务功能,例如文件重命名、文件删除、目录创建、目录删除等。要使用 os 模块,需要先导入该模块,然后调用相关的方法。

表 9.5 列举了 os 模块中关于目录/文件操作的常用函数及其功能。

表 9.5 os 模块中关于目录/文件操作的常用函数及其功能

函数名	函数功能
getcwd()	显示当前的工作目录
chdir(newdir)	改变当前工作目录
listdir(path)	列出指定目录下所有的文件和目录
mkdir(path)	创建单级目录
makedirs(path)	递归地创建多级目录
rmdir(path)	删除单级目录
removedirs(path)	递归地删除多级空目录,从子目录到父目录逐层删除,遇到目录非空则抛出异常
rename(old,new)	将文件或目录 old 重命名为 new
remove(path)	删除文件
stat(file)	获取文件 file 的所有属性
chmod(file)	修改文件权限
system(command)	执行操作系统命令
exec()或 execvp()	启动新进程
osspawnv()	在后台执行程序
exit()	终止当前进程

下面介绍 os 模块中主要函数的使用方法。

1．getcwd()

功能：显示当前工作目录。例如：

```
>>> os.getcwd()
'C:\\Users\\User\\AppData\\Local\\Programs\\Python\\Python35-32'
```

2．chdir(newdir)

功能：改变当前工作目录。例如：

```
>>> os.chdir("e:\\")
>>> os.getcwd()
'e:\\'
```

3．listdir(path)

功能：列出指定目录下所有的文件和目录，参数 path 用于指定列举的目录。例如：

```
>>> os.listdir("c:\\")
['$360Section','$Recycle.Bin','1.dat','360SANDBOX','Documents and Settings','hiberfil.sys','Intel','kjcg8','LibAntiPrtSc_ERROR.log','LibAntiPrtSc_INFORMATION.log','MSOCache','pagefile.sys','PerfLogs','Program Files','Program Files (x86)','ProgramData','Python27','Recovery','System Volume Information','Users','Windows']
```

4．mkdir(path)

功能：创建单级目录。如果要创建的目录存在，则抛出 FileExistsError 异常(异常处理的相关内容将在第 10 章介绍)。例如：

```
>>> os.mkdir("Python")
>>> os.mkdir("Python")
Traceback (most recent call last):
  File "<pyshell#7>",line 1,in <module>
    os.mkdir("Python")
FileExistsError: [WinError 183]当文件已存在时,无法创建该文件。: 'Python'
makedirs(path)
```

5．makedirs()

功能：递归地创建多级目录。如果目录存在，则抛出异常。例如：

```
>>> os.makedirs(r"e:\\aa\\bb\\cc")
```

创建目录结果如图 9.4 所示。



图 9.4　makedirs()函数结果

6．rmdir(path)

功能：删除单级目录。如果指定目录非空，则抛出 PermissionError 异常。例如：

```
>>> os.rmdir("e:\\Python")
>>> os.rmdir("e:\\Python")
Traceback (most recent call last):
  File "<pyshell#10>",line 1,in <module>
    os.rmdir("e:\\Python")
FileNotFoundError: [WinError 2]系统找不到指定的文件。: 'e:\\Python'
>>> os.rmdir("e:\\")
Traceback (most recent call last):
  File "<pyshell#11>",line 1,in <module>
    os.rmdir("e:\\")
PermissionError: [WinError 32]另一个程序正在使用此文件,进程无法访问。: 'e:\\'
```

7. removedirs(path)

功能：递归地删除多级空目录，从子目录到父目录逐层删除，遇到目录非空则抛出异常。例如：

```
>>> os.chdir("e:\\")
>>> os.getcwd()
'e:\\'
>>> os.removedirs(r"aa\bb\cc")
```

8. rename(old，new)

功能：将文件或目录 old 重命名为 new。例如：

```
>>> os.rename("a.txt","b,txt")      #将文件 a.txt 重命名为 b.txt
```

9. remove(path)

功能：删除文件。如果文件不存在，抛出异常。例如：

```
>>> os.remove("b.txt")              #删除 b.txt 文件
>>> os.remove("b.txt")
Traceback (most recent call last):
  File "<pyshell#13>",line 1,in <module>
    os.remove("b.txt")
FileNotFoundError: [WinError 2]系统找不到指定的文件。: 'b.txt'
```

10. stat(file)

功能：获取文件 file 的所有属性。例如：

```
>>> os.stat("string1.txt")
```

```
os.stat_result(st_mode=33206,st_ino=281474976749938,st_dev=2391163256,st_ nlink=1,st_uid=0,st_gid=0,st_size=9,st_atime=1491662692,st_mtime= 1491662692,st_ctime=1491662692)
```

9.5.2 os.path 模块

Python 中的 os. path 模块主要用于针对路径的操作。

表 9.6 列举了 os. path 模块中常用的函数及其功能。

表 9.6 os. path 模块中常用的函数及其功能

函 数 名	函 数 功 能
split(path)	分离文件名与路径
splitext(path)	分离文件名与扩展名,返回(f_path,f_name)元组
abspath(path)	获得文件的绝对路径
dirname(path)	去掉文件名,只返回目录路径
getsize(file)	获得指定文件的大小,返回值以字节为单位
getatime(file)	返回指定文件最近的访问时间
getctime(file)	返回指定文件的创建时间
getmtime(file)	返回指定文件最新的修改时间
basename(path)	去掉目录路径,只返回路径中的文件名
exists(path)	判断文件或者目录是否存在
islink(path)	判断指定路径是否绝对路径
isfile(path)	判断指定路径是否存在且是一个文件
isdir(path)	判断指定路径是否存在且是一个目录
isabs(path)	判断指定路径是否存在且是一个
walk(path)	搜索目录下的所有文件

下面介绍 os. path 模块中主要函数的使用方法。

1. split(path)

功能:分离文件名与路径,返回(f_path,f_name)元组。如果 path 中是一个目录和文件名,则输出路径和文件名全部是路径;如果 path 中是一个目录名,则输出路径和空文件名。例如:

```
>>> os.path.split('e:\\program\\soft\\python\\')
('e:\\program\\soft\\python','')
>>> os.path.split('e:\\program\\soft\\python')
('e:\\program\\soft','python')
```

2. splitext(path)

功能：分离文件名与扩展名。

```
>>> os.path.splitext('e:\\program\\soft\\python\\prime.py')
('e:\\program\\soft\\python\\prime','.py')
```

3. abspath(path)

功能：获得文件的绝对路径。

```
>>> os.path.abspath('prime.py')
'C:\\Users\\User\\AppData\\Local\\Programs\\Python\\Python35-32\\prime.py'
```

4. getsize(file)

功能：获得指定文件的大小，返回值以字节为单位。例如：

```
>>> os.chdir(r"e:\\")
>>> os.path.getsize('e:\\string1.txt')
9
```

5. getatime(file)

功能：返回指定文件最近的访问时间。返回值是浮点型秒数，可以使用 time 模块的 gmtime()或 localtime()函数换算。例如：

```
>>> os.path.getatime('e:\\string1.txt')
1491662692.6680775
>>> import time
>>> time.localtime(os.path.getatime('e:\\string1.txt'))
time.struct_time(tm_year=2017,tm_mon=4,tm_mday=8,tm_hour=22,tm_min=44,tm_sec=52,tm_wday=5,tm_yday=
    98,tm_isdst=0)
```

6. exists(path)

功能：判断文件或者目录是否存在，返回值为 True 或 False。例如：

```
>>> os.path.exists("prime.py")
True
```

9.6 文件应用举例

【例 9.7】 有两个磁盘文件 string1.txt 和 string2.txt，各存放一行字母，读取这两个文件中的信息并合并，然后再写到一个新的磁盘文件 string.txt 中。

程序如下：

```
fp=open("e:\\string1.txt","rt")
```

```
print("读取到文件 string1 的内容为:")
string1=fp.read()
print(string1)
fp.close()
fp=open("e:\\string2.txt","rt");
print("读取到文件 string1 的内容为:")
string2=fp.read()
print(string2)
fp.close()

string=string1+string2
print("合并后字符串内容为:\n",string)

fp=open("e:\\string.txt","wt");
fp.write(string)                    #将字符串 string 的内容写到 fp 所指的文件中
print("已将该内容写入文件 string.txt 中!");
fp.close()
```

【例 9.8】 输入文件名,生成文件,并生成随机数写入该文件,再读取文件内容。

程序如下:

```
import random
filename=input("请输入文件名:")
line=""
fp=open(filename,"w")               #以写方式打开文件
for i in range(100):
  line+='编号:'+str(random.random())+'\n'
  fp.write(line)                    #将字符串 line 写入文件
fp.close()
fp=open(filename,"r")               #再次以读方式打开文件
lines=fp.read()
for s in lines.split('\n'):         #读取文件并按行输出
  print(s)
fp.close()
```

【例 9.9】 将文件夹下所有图片名称加上'_Python'。

程序如下:

```
import re
import os
import time

def change_name(path):
    global i
    if not os.path.isdir(path) and not os.path.isfile(path):
        return False
```

```
    if os.path.isfile(path):
        file_path = os.path.split(path)  #分割出目录与文件
        lists = file_path[1].split('.')  #分割出文件与文件扩展名
        file_ext = lists[-1]
        img_ext = ['bmp','jpeg','gif','psd','png','jpg']
        if file_ext in img_ext:
            os.rename(path,file_path[0]+'/'+lists[0]+'_Python.'+file_ext)
            i+=1
    elif os.path.isdir(path):
        for x in os.listdir(path):
            change_name(os.path.join(path,x))

#测试代码
img_dir="f:\\qwer"
img_dir=img_dir.replace('\\','/')
start=time.time()
i=0
change_name(img_dir)
c=time.time()-start
print('程序运行耗时:%0.2f'%(c))
print('总共处理了%s张图片'%(i))
```

习　　题

1. 编写一个比较两个文件内容是否相同的程序。若相同,显示 compare ok;否则,显示 not equal。

2. 从键盘上输入一行字符,将其中的大写字母全部转换为小写字母,然后输出到一个磁盘文件中保存。

3. 先建立一个文本文件,然后将文件中的内容读出,并将大写字母转换为小写字母,并重新写回文件。

4. 将字符串"Python Program"写入文件,查看文件的字节数。

5. 递归地显示当前目录下所有的目录及文件。

第10章 异常处理

异常(exception)是程序运行过程中发生的事件。该事件可以中断程序指令的正常执行流程,是一种常见的运行错误。例如,除法运算时除数为0、访问序列时下标越界、要打开的文件不存在和网络异常等。如果这些事件得不到正确的处理,将会导致程序终止运行。而合理地使用异常处理结果可以使得程序更加健壮、具有更强的容错性,不会因为用户不小心的错误输入或其他运行时的原因而造成程序终止运行,也可以使用异常处理结构为用户提供更加友好的提示。

10.1 异　　常

异常代表了应用程序的某种反常状态,通常这种应用程序中出现的异常会产生某些类型的错误。程序中的错误通常分为三种。

(1) 语法错误。是指程序中含有不符合语法规定的语句。例如,关键字或符号书写错误(将数组元素引用写成a(2)等)、使用了未定义的变量、括号不配对等。含有语法错误的程序是不能通过编译的,因此程序将不能运行。

(2) 逻辑错误。是指程序中没有语法错误,可以通过编译、连接生成可执行程序,但程序运行的结果与预期不相符的错误。例如,整型变量的取值超出了有效的取值范围、在scanf函数遗漏了取地址运算符&、数组元素引用中下标越界、在应当使用复合语句时没有使用复合语句等。由于含有逻辑错误的程序仍然可以运行,因此这是一种较难发现、较难调试的程序错误,在程序设计、调试中应特别注意。

(3) 系统错误。是指程序没有语法错误和逻辑错误,但程序的正常运行依赖于某些外部条件的存在,如果这些外部条件缺失,则程序将不能运行。例如,折半查找法是在已经排序的数组上进行的,但实际的数据并没有进行排序;程序中需要打开一个已经存在的文件,但这个文件由于其他原因丢失等。

即使语句没有语法错误,在运行程序时也可能发生错误。运行时检测到的错误称为例外,这种错误不一定是致命的。多数例外不能被程序处理,而只是会产生错误信息。例如:

```
>>> 10* (3/0)
Traceback (most recent call last):
  File "<pyshell# 0>",line 1,in <module>
    10* (3/0)
```

```
ZeroDivisionError: division by zero
>>> 5+ '5'
Traceback (most recent call last):
  File "<pyshell# 3>",line 1,in <module>
    5+ '5'
TypeError: unsupported operand type(s) for+ : 'int' and 'str'
>>> a= add(b)
Traceback (most recent call last):
  File "<pyshell# 4>",line 1,in <module>
    a= add(b)
NameError: name 'add' is not defined
```

错误信息的最后一行显示错误的类型,其余部分是错误的细节,其解释依赖于例外类型。例外有不同的类型,作为错误信息的一部分显示。上例中错误的类型有ZeroDivisionError、TypeError 和 NameError。Python 标准异常如表 10.1 所示。

表 10.1　Python 标准异常

异常名称	描　　述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入 Ctrl+C)
GeneratorExit	生成器(generator)发生异常来通知退出
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
StandardError	所有的内建标准异常的基类
BufferError	无法执行缓冲区相关操作时的错误
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零(所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
VMSError	发生 VMS 特定错误时引发
EOFError	没有内建输入,到达 EOF 标记

续表

异常名称	描述
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误
NameError	未声明/初始化对象(没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告
FutureWarning	关于构造将来语义会有改变的警告
ImportWarning	导入模块时出现问题的警告
UnicodeWarning	Unicode 文本中问题的警告
BytesWarning	可疑字节的警告

在 Python 中,各种异常错误都是类,所有的错误类型都继承于 BaseException。常见异常类型的继承关系如图 10.1 所示。

```
BaseException
+--SystemExit
+--KeyboardInterrupt
+--GeneratorExit
+--Exception
      +--StopIteration
      +--StandardError
      |    +--BufferError
      |    +--ArithmeticError
      |    |    +--FloatingPointError
      |    |    +--OverflowError
      |    |    +--ZeroDivisionError
      |    +--AssertionError
      |    +--AttributeError
      |    +--EnvironmentError
      |    |    +--IOError
      |    |    +--OSError
      |    |         +--WindowsError (Windows)
      |    |         +--VMSError (VMS)
      |    +--EOFError
      |    +--ImportError
      |    +--LookupError
      |    |    +--IndexError
      |    |    +--KeyError
      |    +--MemoryError
      |    +--NameError
      |    |    +--UnboundLocalError
      |    +--ReferenceError
      |    +--RuntimeError
      |    |    +--NotImplementedError
      |    +--SyntaxError
      |    |    +--IndentationError
      |    |         +--TabError
      |    +--SystemError
      |    +--TypeError
      |    +--ValueError
      |        +--UnicodeError
      |              +--UnicodeDecodeError
      |              +--UnicodeEncodeError
      |              +--UnicodeTranslateError
      +--Warning
          +--DeprecationWarning
          +--PendingDeprecationWarning
          +--RuntimeWarning
          +--SyntaxWarning
          +--UserWarning
          +--FutureWarning
  +--ImportWarning
  +--UnicodeWarning
  +--BytesWarning
```

图 10.1 常见异常类继承关系

10.2　Python 中异常处理结构

Python 中捕捉异常可以使用 try…except 语句。try…except 语句用来检测 try 语句块中的错误,从而让 except 语句捕获异常信息并处理。

10.2.1　简单形式的 try…except 语句

简单形式的 try…except 语句一般形式为:

```
try:
      语句块
except:
      异常处理语句块
```

其处理过程是:执行 try 中的语句块,如果执行正常,在语句块执行结束后转向 try…except 语句之后的下一条语句;如果引发异常,则转向异常处理语句块,执行结束后转向 try…except 语句之后的下一条语句。

【例 10.1】　猜数字游戏。

分析:可以随机产生一个整数,从键盘输入整数,输入的数字如果大于随机产生的整数,则输出“guess bigger”,继续输入;输入的数字如果小于随机产生的整数,则输出“guess smaller”,继续输入;输入的数字如果等于随机产生的整数,则输出“You guess correct. Game over!”,程序运行结束。在输入的过程中,如果输入的不是整数,则引发异常,要处理异常。

程序如下:

```
import random
num=random.randint(1,10)
while True:
  try:
    guess=int(input("Enter 1~10:"))
  except:
    print("Input error,Please Enter number 1~10:")
    continue
  if guess>num:
    print("guess bigger")
  elif guess<num:
    print("guess smaller")
  else:
    print("You guess correct.Game over!")
    break
```

程序运行结果：

```
Enter 1~10:5
guess bigger
Enter 1~10:2
guess bigger
Enter 1~10:1
You guess correct.Game over!
```

以上的运行中，输入的都是整数，因此没有引发异常；如果输入其他类型的数据，则会引发异常。异常发生的程序运行结果如下：

```
Enter 1~10:1
guess smaller
Enter 1~10:'a'
Input error,Please Enter number 1~10:
Enter 1~10:2.0
Input error,Please Enter number 1~10:
Enter 1~10:5
guess bigger
Enter 1~10:4
You guess correct.Game over!
```

在以上 try…except 语句的一般形式中，except 之后也可以用来处理指定特定错误类型的异常。

【例 10.2】 除数为 0 的异常处理。

程序如下：

```
numbers=[0.33,2.5,0,100]
for x in numbers:
    print(x)
    try:
        print(1.0/x)
    except ZeroDivisionError:
        print("除数不能为零")
```

程序运行结果：

```
0.33
3.0303030303030303
2.5
0.4
0
除数不能为零
100
0.01
```

在该例中,try 语句执行过程如下。

(1) 执行 try 子句(在 try 和 except 之间的语句),该例中 try 子句是“print(1.0/x)”。

(2) 如果没有发生例外,跳过 except 子句,try 语句运行完毕。

(3) 如果在 try 子句中发生了例外错误,同时例外错误匹配 except 后指定的例外名,则跳过 try 子句剩下的部分,执行 except 子句,然后继续执行 try 语句后面的程序。

(4)如果在 try 子句中发生了例外错误,但是例外错误和 except 后指定的例外名不匹配,则此例外被传给外层的 try 语句。如果没有找到匹配的处理程序,则此例外被称作未处理例外,程序停止执行,显示错误信息。

10.2.2 带有多个 except 的 try 语句

在实际开发过程中,同一段代码可能会抛出多个异常,需要针对不同的异常类型进行相应的处理。为了支持对多个异常的处理,Python 提供了带有多个 except 的异常处理结构,类似于多路分支选择结构。其一般形式为:

```
try:
    语句块
except 异常类型 1:
    异常处理语句块 1
except 异常类型 2:
    异常处理语句块 2
    …
except 异常类型 n:
    异常处理语句块 n
except:
    异常处理语句块
else:
    语句块
```

其处理过程是:执行 try 中的语句块,如果执行正常,在语句块执行结束后转向 try…except 语句之后的下一条语句;如果引发异常,则系统依次检查各个 except 子句,将所发生的异常与 except 之后的异常类型进行匹配。如果找到相匹配的错误类型,则执行相应的异常处理语句块;如果找不到,则执行最后一个 except 子句中的默认异常处理语句块。如果执行 try 语句块是没有发生异常,Python 系统则执行 else 语句后的语句块。异常处理结束后转向 try…except 语句之后的下一条语句。注意:上面的表示形式中最后一个 except 子句和 else 子句都是可选的。

【例 10.3】 带有多个 except 的异常处理。

程序如下:

```
try:
  x= input("请输入被除数:")
  y= input("请输入除数:")
  a= int(x)/float(y) * z
```

```
except ZeroDivisionError:
  print("除数不能为零")
except NameError:
  print("变量不存在")
else:
  print(x,"/",y,"=",z)
```

程序运行结果：

```
请输入被除数：3
请输入除数：4
变量不存在
```

再次运行程序，结果如下：

```
请输入被除数：3
请输入除数：0
除数不能为零
```

【例 10.4】 字符串输出。

程序如下：

```
a_list=["apple","pear","banana","peach"]
while True:
    n=int(input("请输入要输出的字符串的序号:"))
    try:
      print(a_list[n])
    except:
      print("列表元素的下标越界或格式不正确")
    else:
      break
```

程序运行结果：

```
请输入要输出的字符串的序号：1
Pear
```

再次运行程序，结果如下：

```
请输入要输出的字符串的序号：5
列表元素的下标越界或格式不正确
请输入要输出的字符串的序号：3
peach
```

在该例中，如果 try 中的代码没有抛出任何异常，则执行 else 块中的代码 break，也就是循环结束。

可以使用一个 except 捕获多个异常，将多个异常类型写在括号里用逗号隔开，形成

一个元组,并且共用同一段异常处理语句块。

10.2.3　try…except…finally 语句结构

try…except…finally 语句的一般形式为:

```
try:
    语句块:
except:
    异常处理语句块
finally:
    语句块
```

其处理过程是:执行 try 中的语句块,如果执行正常,在 try 语句块执行结束后执行 finally 语句块,然后再转向 try…except 语句之后的下一条语句;如果引发异常,则转向 except 异常处理语句块,该语句块执行结束后执行 finally 语句块。也就是说无论是否检测到异常,都会执行 finally 代码,因此一般会把一些清理工作,例如关闭文件或者释放资源等,写到 finally 语句块中。

【例 10.5】　文件读取异常处理。

程序如下:

```
try:
  fp=open("test.txt","r")
  ss=fp.read()
  print("Read the contents:",ss)
except IOError:
  print("IOError")
finally:
  print("close file!")
  fp.close()
```

如果在当前目录下不存在 test.txt 文件,则程序运行结果如下:

```
IOError
close file!
```

由于文件不存在,执行 try 语句时产生异常,执行 except 中的异常语句处理块,最后再执行 finally 语句块。

如果在当前目录下创建 test.txt 文件,并将字符串"abcdefg"写入文件中,则程序运行结果如下:

```
Read the contents: abcdefg
close file!
```

此时执行 try 语句块时没有产生异常,该 try 语句块执行完成后执行 finally 语句块。

10.3 自定义异常

Python 中允许自定义异常，用于描述 Python 中没有涉及的异常情况。Python 中异常是类，要自定义异常，就必须首先创建其中一个异常类（见图 10.1）的子类，通过继承，将异常类的所有基本特点保留下来。新创建的异常类将提供方法，用户生成的类提供独有的错误。

定义自己的异常类一般通过直接或间接的方式继承自 Exception 类，初始化时同时使用 Exception 类的__init__()方法。使用 raise exceptiontype(arg…)语法引发自己定义的异常，直接生成该异常类的一个实例并抛出该异常。在捕获异常时使用 except exceptiontype as var 语法获取异常实例 var，从而可以在后续的处理中访问该异常实例的属性。

下面是用户自定义的与 RuntimeError 相关的异常实例。实例中创建了一个类，基类为 RuntimeError，用于在异常触发时输出更多的信息。在 try 语句块中，用户自定义异常后执行 except 块语句，变量 e 用于创建 Networkerror 类的实例。

```
class Networkerror(RuntimeError):
  def __init__(self,arg):
    self.args=arg
try:
  raise Networkerror("myexception")
except Networkerror as e:
  print(e.args)
```

以上程序运行结果如下：

```
('m', 'y', 'e', 'x', 'c', 'e', 'p', 't', 'i', 'o', 'n')
```

10.4 断言与上下文管理

断言与上下文管理是两种特殊的异常处理方式，在形式上比 try 语句要简单一些，能够满足简单的异常处理，也可以与标准的异常处理结构 try 语句结合使用。

10.4.1 断言

断言的作用是帮助调试程序，以保证程序的正确性。

Python 中使用 assert 语句可以声明断言，其一般形式为：

```
assertexpression[,reason]
```

其处理过程是：该语句有一个或两个参数，第二个参数为可选项。第一个参数

expression 是一个逻辑值，执行时首先判断表达式 expression 的值，如果该值为 True，什么都不做；如果该值为 False，断言不通过，则抛出异常。第二个参数是错误的描述，即断言失败时输出的信息。

【例 10.6】 判断素数的断言处理。

程序如下：

```
def isPrime(n):
    assert n>=2
    from math import sqrt
    for i in range(2,int(sqrt(n))+1):
        if n%i==0:
            return False
    return True
while True:
  n=int(input("请输入一个整数："))
  flag=isPrime(n)
  if flag==True:
     print("%d是素数"%n)
  else:
     print("%d不是素数"%n)
```

程序运行结果：

```
请输入一个整数：23
23是素数
请输入一个整数：32
32不是素数
请输入一个整数：1
Traceback (most recent call last):
  File "F:/python/ isPrime.py",line 11,in <module>
    flag=isPrime(n)
  File " F:/python/ isPrime.py.py",line 3,in isPrime
    assert n>=2
AssertionError
```

在上例中，assert 异常被捕获，但没有对应的处理语句，因此它使程序终止并产生回溯。

说明：

(1) assert 语句用来声明某个条件是真的，当 assert 语句为假的时候，会引发 AssertionError 错误。例如：

```
>>> assert 1==1
>>> assert 1>1
```

```
Traceback (most recent call last):
  File "<pyshell#2>",line 1,in <module>
    assert 1>1
AssertionError
```

(2) assert 语句与异常处理 try 经常结合使用。

【例 10.7】 AssertionError 异常处理。

程序如下:

```
for i in range(3):
  str=input("entry string:")
  try:
    assert len(str) ==5
    print("string:%s"%str)
  except AssertionError:
    print("Assertion Error")
```

程序运行结果:

```
entry string:hello
string: hello
entry string:assert
Assertion Error
entry string:error
string:error
```

10.4.2 上下文管理

使用 with 语句实现上下文管理功能,用于规定某个对象的使用范围。使用 with 自动关闭资源,可以在代码块执行完毕后还原进入该代码块时的现场,不论何种原因跳出 with 块,不论是否发生异常,总能保证文件被正确关闭,资源被正确释放。它常用于文件操作、网络通信等之类的场合。

with 语句的一般形式为:

```
with context_expression [asvar]:
    with语句块
```

【例 10.8】 with 的应用。

```
with open('test.txt') as f:
   for line in f:
         print(line,end=' ')
```

在上面的例子中,当文件处理完成时,会自动关闭文件。

习　　题

1．程序中的错误通常有哪几种？

2．Python 异常处理结构有哪些？

3．语句 try…except 和 try…finally 有什么不同？

4．编写程序，输入三个数字。用输入的第一个数字除以第二个数字，得到的结果与第二个数字相加。使用异常检查可能出现的错误：IOError、ValueError 和 ZeroDivisionError。

第 11 章 面向对象程序设计

Python 采用面向对象程序设计(Object Oriented Programming,OOP)思想,是真正面向对象的高级动态编程语言支持面向对象的基本功能。本章系统地介绍面向对象程序设计的基本概念和 Python 面向对象程序设计的基本方法,包括类、对象、继承、多态以及对基类对象的覆盖或重写。通过本章的学习,读者能够更多地认识到 Python 面向对象程序设计。

11.1 面向对象程序设计概述

面向对象程序设计是一种计算机编程架构,主要针对大型软件设计而提出。面向对象程序设计是软件工程、结构化程序设计、数据抽象、信息隐藏、知识表示及并行处理等多种理论的积累和发展,使得软件设计更加灵活,更好地支持代码复用和设计复用,使代码更具有可读性和可扩展性。

11.1.1 面向对象的基本概念

面向对象方法是一种集问题分析方法、软件设计方法和人类的思维于一体的贯穿软件系统分析、设计和实现整个过程的程序设计方法。面向对象方法的基本思想是:对问题空间进行自然分割,以更接近人类思维的方式建立问题域模型,以便对客观实体进行结构模拟和行为模拟,从而使设计出的软件尽可能直接地描述现实世界,并限制软件的复杂性,降低软件开发费用,从而构造出模块化的、可重用的、维护方便的软件。面向对象方法中,对象作为描述信息实体的统一概念,把属性和服务融为一体,通过类、对象、服务、消息、继承、封装、多态、联编等概念和机制构造软件系统,并为软件重用和方便维护提供强有力的支持。

1. 对象

现实世界中客观存在的事物称为对象(object)。在现实世界中,对象有两大类:①人们身边存在的一切事物,如一个人、一本书、一座大楼、一棵树等;②人们身边发生的一切事件,如一场篮球比赛、一个到图书馆的借书过程、一次演出等。不同的对象有不同的特征和功能。例如,一个人有姓名、性别、年龄、身高、体重等特征,也具有说话、行走等功能。

现实世界是由一个个这样的对象相互之间的有机联系组成的。

如果把现实世界中的对象进行计算机数字化转换,则这样的对象具有如下特征。

(1) 用一个名称来唯一标识对象。

(2) 用一组状态来描述其特征。

(3) 用一组操作来实现其功能。

2. 类

类(class)是对一组具有相同属性和相同操作对象的抽象。一个类就是对一组相似对象的共同描述,它整体地代表一组对象。类封装了对描述某些现实世界对象的内容和行为所需的数据和操作的抽象,它给出了属于该类的全部对象的抽象定义,包括类的属性、操作和其他性质。对象只是符合某个类定义的一个实体,属于某个类的一个具体对象称为该类的一个实例(instance)。

可以把类看作某些对象的模板(template),抽象地描述了属于该类的全部对象共有的属性和操作。类与对象(实例)的关系是抽象与具体的关系,类是多个对象(实例)的综合抽象,对象(实例)是类的个体实物。例如,在学生信息管理系统中,学生是一个类,它是一个特殊的人的群体。学生类的属性有学号、姓名、性别、年龄等,可能定义了入学注册、选课等操作。张三是一名学生,是一个具体的对象,也是学生类的一个实例。

一个类的构造至少应包括以下方面。

(1) 类的名称。

(2) 属性结构,包括所用的类型、实例变量及操作的定义。

(3) 与其他类的关系,如继承关系等。

(4) 外部操作类的实例的操作界面。

3. 消息

消息(message)是指对象之间在交互中所传递的通信信息。面向对象的封装机制使对象各自独立,各司其职。消息是对象间交互、协同工作的手段,它激发接收对象产生某种服务操作,通过操作的执行来完成相应的服务行为。当一个消息发送给某个对象时,包含有要求接收对象去执行某个服务的信息。接收到消息的对象经过解释,然后予以响应,这种通信机制称作消息传递。发送消息的对象不需要知道接收消息的对象如何对消息予以响应。

通常一个消息由以下三部分组成。

(1) 接收消息的对象。

(2) 消息名。

(3) 零个或多个参数。

4. 封装

在面向对象方法中,对象的属性和方法的实现代码被封装在对象的内部。一个对象就像是一个黑盒子,表示对象状态的属性和服务的实现代码被封装存放在黑盒子里,从外面无法看见,更不能修改。对象向外界提供访问的接口,外界只能通过对象的接口来访问该对象。外界通过对象的接口访问对象称为向该对象发送消息。对象具有的这种封装特性称为封装性(encapsulation)。类是对象封装的工具,对象是封装的实现。

封装的信息隐蔽作用反映了事物的相对独立性,使我们只关心它对外所提供的接口,即它能做什么,而不注意它的内部细节,即怎么提供这些方法。封装的结果使对象以外的

部分不能随意存取对象的内部属性,从而有效避免了外部错误对它的影响,大大减少了差错和排错的难度。另外,当对对象内部进行修改时,由于它只通过少量的服务接口对外提供服务,因此同样减小了内部修改对外部的影响。

5. 继承性

继承性(inheritance)是面向对象程序设计的一个重要特性,它允许在已有类的基础上创建新的类,新类可以从一个或多个已有类中继承函数和数据,而且可以重新定义或加进新的数据和函数,从而新类不但可以共享原有类的属性,同时也具有新的特性,这样就形成类的层次或等级。

可以让一个类拥有全部的属性,还可以通过继承让另一个类继承它的全部属性。被继承的类称为基类或者父类,而继承的类或者说是派生出来的新类称为派生类或者子类。

对于一个派生类,如果只有一个基类,称为单继承。如果同时有多个基类,称为多重继承。单继承可以看成是多重继承的一个最简单特例,而多重继承可以看成是多个单继承的组合。

类的继承具有传递性,即如果类 C 是类 B 的子类,类 B 是类 A 的子类,则类 C 不仅继承类 B 的所有属性和方法,还继承类 A 的所有属性和功能。因此,一个类实际上继承了它所在类层次以上层的全部父类的属性和方法。这样,属于该类的对象不仅具有自己的属性和方法,还具有该类所有父类的属性和方法。

6. 多态性

多态性(polymorphism)一词来源于希腊,从字面上理解,poly(表示多的意思)和morphos(意为形态)即 many forms,是指同一种事物具有多种形态。在自然语言中,多态性是"一词多义",是指相同的动词作用到不同类型的对象上。例如,驾驶摩托车、驾驶汽车、驾驶飞机、驾驶轮船、驾驶火车等这些行为都具有相同的动作——驾驶,但它们各自作用的对象不同,其具体的驾驶动作也不同,但却都表达了同样的一种含义——驾驶交通工具。试想如果用不同的动词来表达"驾驶"这一含义,那将会在使用中产生很多麻烦。

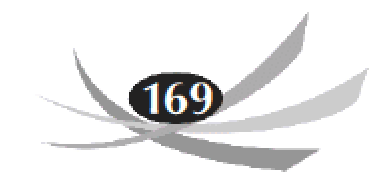
简单地说,多态性是指类中具有相似功能的不同函数使用同一名称,从而使得可以用相同的调用方式达到调用具有不同功能的同名函数的效果。在面向对象程序设计语言中,多态性是指不同对象接收到相同的消息时产生不同的响应动作,即对应相同的函数名,却执行不同的函数体,从而用同样的接口去访问功能不同的函数,实现"一个接口,多种方法"。

11.1.2 从面向过程到面向对象

面向过程(object oriented,OO)的程序设计是一种自上而下的设计方法,Niklaus Wirth 提出计算机系统中一个重要的公式:

程序=数据结构+算法

该公式体现了面向过程程序设计的核心思想,即数据与算法。在面向过程的程序设计方法中,将数据与数据处理过程分开,对程序按照功能分割成一个个小的子函数,逐步分解,将问题拆分为一个个较小的功能模块。面向过程的程序设计是以函数为中心,用函



数作为划分程序的基本单位,数据在其中起到从属的作用。

面向过程程序设计易于理解和掌握,但在处理一些较为复杂的问题时会存在许多问题。面向过程程序设计一般既有定义数据的元素,也有定义操作的元素,即将数据与操作分割,这样不易于维护程序。除此之外,还存在代码复用率低、可扩展性差等缺点。

面向对象程序设计是一种自下而上的程序设计方法,将数据与数据处理当作一个整体,即一个对象。相较于面向过程的程序设计方法,面向对象程序设计有以下优点。

(1) 将数据抽象化,可在外部接口不改变的前提下改变内部实现,避免或减少外部干扰。

(2) 通过继承可大幅度减少冗余代码,降低代码出错率,提高代码利用率与软件可维护性。

(3) 将对象按照同一属性和行为划分为不同的类,可将软件系统分割为若干个相互独立的部分,便于控制软件复杂度。

(4) 以对象为核心,开发人员可从静态(属性)和动态(方法)两个方面考虑问题,更好地设计实现系统。

Python 采用面向对象的程序设计思想,是面向对象的高级动态编程语言,完全支持面向过程的基本功能,包括封装、继承、多态以及对基类方法的覆盖和重写。相较于其他编程语言,Python 中对象的概念更加广泛,不仅可以是某个类的实例化,也可以是任何内容。例如,字典、元组等内置数据类型也同时具有与类完全相似的语法和用法。创建类时用变量形式表示的对象属性成为数据成员或成员属性,用函数形式表示的对象行为称为成员函数或成员方法,成员属性和成员方法统称为类的成员。

11.2　类与对象

11.2.1　类的定义

类和对象是计算机系统世界中重要的两个概念,类是客观事物的抽象,对象是类的实例化。Python 中使用 class 关键字定义类,定义类的一般方法为:

```
class 类名:
  类的内部实现
```

class 关键字后紧跟空格,空格之后是类的名称,类名称后面必须有冒号,然后换行,以空格控制 Python 逻辑关系,最后定义类的内部实现。

类名一般首字母需要大写,当然也可以按照个人习惯。但需要注意整个系统的设计与实现保持风格一致。

【例 11.1】 类的定义。

程序如下:

```
class Cat:
    def describe (self):
```

```
        print (' This is a cat ')
```

本例中,Cat 类只有一个方法 describe(),类方法中至少需要一个参数 self,self 表示实例化对象本身。

注意:类的所有实例方法必须有一个参数为 self,self 代表将来要创建的对象本身,并且必须是第一个形参。在类的实例方法中访问实例属性需要以 self 作为前缀,在外部通过类名调用对象方法同样需要以 self 作为参数传值,而在外部通过对象名调用对象方法时不需要传递该参数。

实际应用中,在类中定义实例方法时第一个参数并不必须名为 self,也可以由开发人员自定义。

【例 11.2】 类的定义 2。

程序如下:

```
class Dog:
    def __init__(this,d):
        this.value = d
    def show(this):
        print(this.value)
    d= Dog (23)
    d.show()
```

程序运行结果:

```
23
```

11.2.2 对象的创建和使用

类是抽象的,创建类之后,要使用类定义的功能,必须将其实例化,即创建类的对象。一般形式为:

对象名=类名(参数列表)

创建对象后,可通过“对象名. 成员”的方式访问其中的数据成员或成员方法。

【例 11.3】 对例 11.1 中定义的类进行对象的创建与使用。

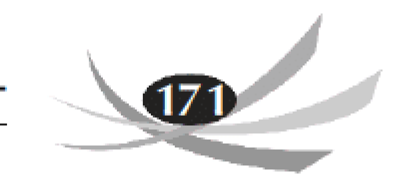
程序如下:

```
cat = Cat()                    #创建对象
cat. describe()                #调用成员方法
```

程序运行结果:

```
This is a cat
```

Python 提供一种内测函数 isinstance()来判断一个对象是否是已知类的实例,其语法如下:




```
isinstance(object,classinfo)
```

其中,第一个参数(object)为对象,第二个参数(classinfo)为类名,返回值为布尔型(True 或 False)。

【例 11.4】 内置函数 isinstance()。

程序如下:

```
>>isinstance (cat,Cat)
```

程序运行结果:

```
True
```

程序如下:

```
>>isinstance (cat,str)
```

程序运行结果:

```
False
```

11.3　属性与方法

11.3.1　实例属性

在 Python 中,属性包括实例属性和类属性两种。实例属性一般是指在构造函数 __init__() 中定义的,定义和使用时必须以 self 为前缀。

Python 中类的构造函数__init__()用来初始化属性,在创建对象时自动执行。构造函数属于对象,每个对象都有属于自己的构造函数。若开发人员未编写构造函数,Python 将提供一个默认的构造函数。

【例 11.5】 实例属性。

程序如下:

```
>>class cat:
>>def __init__(self,s):
>>this.name=s                          #定义实例属性
```

构造函数所对应的即析构函数,Python 中的析构函数是__del__(),用来释放对象所占用的空间资源,在 Python 回收对象空间资源之前自动执行。同样,析构函数属于对象,对象都会有自己的析构函数,若开发人员未定义析构函数,Python 将提供一个默认的析构函数。

注意: __init__中"__"是两个下画线,中间没有空格。

11.3.2　类属性

类属性属于类,是在类中所有方法之外定义的数据成员,可通过类名或对象名访问。

【例 11.6】 类属性定义与使用。

程序如下：

```
class Cat:
    size = ' small '                    #定义类属性
    def __init__(self,s):
        self.name = s                   #定义实例属性
cat1 = Cat('mi')
cat2 = Cat('mao')
print(cat1.name,Cat.size)
```

程序运行结果：

```
mi small
```

在类的方法中可以调用类本身的方法，也可访问类属性及实例属性。值得注意的是，Python 可以动态地为类和对象增加成员，这点是与其他面向对象语言不同的，也是 Python 动态类型的重要特点。

【例 11.7】 动态增加成员。

程序如下：

```
Cat.size = 'big'                        #修改类属性
Cat.price = 1000                        #增加类属性
Cat.name = 'maomi'                      #修改实例属性
```

Python 成员有私有成员和公有成员，若属性名以两个下画线“__”（中间无空格）开头，则该属性为私有属性。私有属性在类的外部不能直接访问，需通过调用对象的公有成员方法或 Python 提供的特殊方式来访问。Python 为访问私有成员所提供的特殊方式用于测试和调试程序，一般不建议使用。该方法如下：

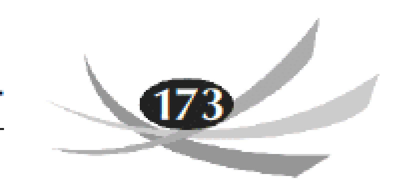
对象名._类名+私有成员

公有属性是公开使用的，既可以在类的内部使用，也可以在类的外部程序中使用。

【例 11.8】 公有成员和私有成员。

程序如下：

```
class Animal:
    def __init__(self):
        self.name= 'cat'                #定义公有成员
        self.__color= 'white'           #定义私有成员
    def setValue(self,n2,c2):
        self.name= n2                   #类的内部使用公有成员
        self.__color= c2                #类的内部访问私有成员
a= Animal()                             #创建对象
print(a.name)                           #外部访问公有成员
print(_a.__color)                       #外部特殊方式访问私有成员
```



程序运行结果：

```
cat
white
```

注意：Python 中不存在严格意义上的私有成员。

11.3.3　对象方法

类中定义的方法可大致分为三类：私有方法、公有方法和静态方法。私有方法和公有方法属于对象，每个对象都有自己的公有方法和私有方法，这两类方法可访问属于类和对象的成员。公有方法通过对象名直接调用，私有方法以两个下画线“__”(无空格)开始，不能通过对象名直接访问，只能在属于对象的方法中调用或在外部通过 Python 提供的特殊方法调用。静态方法可通过类名和对象名调用，但不能直接访问属于对象的成员，只能访问属于类的成员。

【例 11.9】　公有方法、私有方法和静态方法的定义和调用。

程序如下：

```
class Animal:
    specie='cat'
    def __init__(self):
        self.__name='mao'              #定义和设置私有成员
        self.__color='black'
    def __outPutName(self):            #定义私有函数
        print(self.__name)
    def __outPutColor(self):           #定义私有函数
        print(self.__color)
    def outPut(self):                  #定义公有函数
        outPutName()                   #调用私有方法
        outPutColor()
@staticmethod                          #定义静态方法
def getSpecie():
        return Animal.specie           #调用类属性
@staticmethod
    def setSpecie(s):
        Animal.specie=s
#主程序
cat =Animal()
cat.outPut()                           #调用公有方法
print(Animal.getSpecie())              #调用静态方法
Animal.setSpecie('dog')                #调用静态方法
print(Animal.getSpecie())
```

程序运行结果：

```
mao
black
cat
dog
```

11.4 继承和多态

11.4.1 继承

在面向对象程序设计中，当我们定义一个类时，可通过从已有的类中继承实现。新定义的类称为子类或派生类，而被继承的类称为基类、父类或者超类。继承的方式如下：

```
[语法]  class <父类名> (object):
            <父类内部实现>
        class <子类名> (<父类名>):
            <子类内部实现>
```

其中，基类必须继承于 object，否则派生类将无法使用 super()等函数。

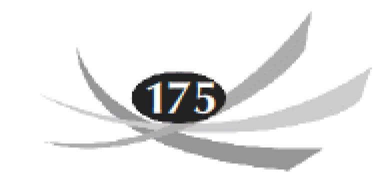
派生类可以继承父类的公有成员，但不能继承父类的私有成员。派生类可通过内置函数 super()调用基类方法或通过以下方式调用：

```
[语法]  基类名.方法名()
```

【例 11.10】 继承的实现。

程序如下：

```
class Animal (object):              #定义基类
    size='small'
    def__init__(self):              #基类构造函数
        self.color='white'
        print ('superClass: init of animal')
    def outPut (self):              #基类公有函数
        print (self.size)
class Dog (Animal):                 #子类 Dog,继承于 Animal 类
    def __init__(self):             #子类构造函数
        self.name='dog'
        print ('subClass: init of dog')
    def run(self):                  #子类方法
        print (Dog.size,self.color,self.name)
        Animal.outPut(self)         #通过父类名调用父类构造函数(方法一)
class Cat (Animal):                 #子类 Cat,继承于 Animal 类
    def __init__(self):             #子类构造函数
        self.name='cat'
        print ('subClass: init of cat')
    def run(self):                  #子类方法
```




```
        print(Cat.size,self.color,self.name)
        super(Cat,self).__init__()  #调用父类构造函数(方式二)
        super().outPut()            #调用父类构造函数(方式三)
#主程序
a=Animal()
a.outPut()
dog=Dog()
dog.size='mid'
dog.color='black'
dog.run()
cat=Cat()
cat.name='maomi'
cat.run()
```

程序运行结果：

```
superClass: init of animal
small
subClass: init of dog
small black dog
mid
subClass: init of cat
superClass: init of animal
small white maomi
small
```

11.4.2　多重继承

Python 支持多重继承，若父类中有相同的方法名，子类在调用过程中并没有指定父类，则子类从左向右按照一定的访问序列逐一访问父类函数，保证每个父类函数仅被调用一次。

【例 11.11】　多重继承应用。

程序如下：

```
class A(object):                    #父类 A
    def __init__(self):
        print('start A')
        print('end A')
    def fun1(self):                 #父类函数
        print('a_fun1')
class B(A):                         #类 B继承于父类 A
    def __init__(self):
        print('start B')
        super(B,self).__init__()
        print('end B')
```

```
    def fun2(self):
        print('b_fun2')
class C(A):                        #类 C继承于父类 A
    def __init__(self):
        print('start C')
        super(C,self).__init__()
        print('end C')
    def fun1(self):                #重写父类函数
        print('c_fun1')
class D(B,C):                      #类 D同时继承于类 B、类 C
    def __init__(self):
        print('start D')
        super(D,self).__init__()
        print('end D')
#主程序
d=D()
d.fun1()
```

程序运行结果:

```
start D
start B
start C
start A
end A
end C
end B
end D
c_fun1
```

说明:

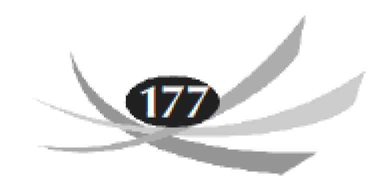
(1) 多重继承访问顺序按照 C3 算法生成 MRO(method resolution order,方法解释顺序)访问序列。例 11.11 中,MRO 访问序列为{D, B, C, A},类 D 中没有 fun1()函数,则按照序列访问父类,首先访问类 B,没有 fun1()函数,然后访问类 C,存在该函数,则调用父类 C 的 fun1()函数。

(2) Python 提供关键字 pass,类似于空语句,可在类、函数定义和选择结构等程序中使用。

(3) Python 提供两种访问父类函数的方法:super()函数调用和父类名调用。在多重继承程序设计中,需注意这两种方法不可混合使用,否则有可能会导致访问序列紊乱。

11.4.3 多态

多态是指不同对象对同一消息做出的不同反应,即"一个接口,不同实现"。按照实现方式,多态可分为编译时多态和运行时多态。编译时多态是指程序在运行前,可根据函数参数不同确定所需调用的函数,运行时多态是指在函数名和函数参数均一致,在程序运行前并不能确定调用的函数。Python 的多态与其他语言不同,Python 变量属于弱类型,定



义变量可以不指明变量类型，并且 Python 语言是一种解释型语言，不需要预编译。因此 Python 语言仅存在运行时多态，程序运行时根据参数类型确定所调用的函数。

【例 11.12】 多态的应用。

程序如下：

```
class A(object):
    def run(self):
        print('this is A')
classB(A):
    def run(self):
        print('this is B')
    classC(A):
    def run(self):
        print('this is C')
#主程序
b=B()
b.run()
c=C()
c.run()
```

程序运行结果：

```
this is B
this is C
```

有了继承才能有多态，在调用实例方法时可以不考虑该方法属于哪个类，将其当作父类对象处理。

11.5　面向对象程序设计举例

【例 11.13】 已知序列 a，求解所有元素的和与所有元素的积。

程序如下：

```
class ListArr:
    def __init__(self):
        self.sum=0
        self.pro=1
    def add(self,l):
        for item in l:
            self.sum+=item
    def product(self,l):
        for item in l:
            self.pro*=item
a=[12,32,63,54]
```

```
l=ListArr()
l.add(a)
l.product(a)
print(l.sum)
print(l.pro)
```

程序运行结果：

```
10
24
```

对 ListArr 类进行操作，首先实例化 ListArr 对象，构建对象时初始化该对象实例属性 sum 和 pro。ListArr 类包含两个方法，即求和函数 add()和求积函数 product()。实例化 ListArr 类的同时，由构建函数初始化类的 sum 属性和 pro 属性，通过调用 add()函数对列表求和，调用 product()函数求列表每个元素的积。

【例 11.14】 随机产生 10 个数的列表，对该列表进行选择排序。

程序如下：

```
import random
class OrderList:
    def __init__(self):
        self.arr = []
        self.num = 0
    def getList(self):
        for i in range(10):
            self.arr.append(random.randint(1,100))
            self.num+ = 1
    def selectSort(self):
            for i in range(0,self.num- 1):
                for j in range(i+ 1,self.num):
                    if self.arr[i]> self.arr[j]:
                        self.arr[i],self.arr[j] = self.arr[j],self.arr[i]
lst=OrderList()
lst.getList()
print("before:",lst.arr)
lst.selectSort()
print("after:",lst.arr)
```

程序运行结果：

```
before: [99,61,44,68,35,87,60,73,3,8]
after: [3,8,35,44,60,61,68,73,87,99]
```

OrderList 类中定义了 getList()方法对象，在实例化对象后，对该类的 arr 属性进行赋值操作，随机产生 10 个数作为 arr 列表元素，调用 selectSort()函数对列表进行选择排



序。选择排序的思想是在未排序的序列中找到最小元素，存放到序列起始位置，再从剩下未排序序列选择最小元素，存放到已排序序列末尾，以此类推，直到所有元素均排序完毕。

【例 11.15】 创建一个学校成员类，登记成员名称，统计总人数。教师类与学生类分别继承学校成员类，登记教师所带班级与学生成绩，每创建一个对象学校总人数加一，删除一个对象则减一。

程序如下：

```
class SchoolMember:
    #总人数,这个是类的变量
    sum_member = 0

    #__init__构造函数在类的对象被创建时执行
    def __init__(self,name):
        self.name = name
        SchoolMember.sum_member+ = 1
        print("学校新加入一个成员：%s" % self.name)
        print("学校共有%d人" % SchoolMember.sum_member)

    #自我介绍
    def say_hello(self):
        print("大家好,我叫%s" % self.name)

    #__del__构造函数在对象不使用的时候运行
        def __del__(self):
        SchoolMember.sum_member- = 1
        print("%s离开了,学校还有%d人" % (self.name,SchoolMember.sum_ member))

#教师类继承学校成员类
class Teacher(SchoolMember):
    def __init__(self,name,grade):
        SchoolMember.__init__(self,name)
        self.grade = grade

    def say_hello(self):
        SchoolMember.say_hello(self)
        print("我是老师,我带的班级是%s班" % self.grade)

    def __del__(self):
        SchoolMember.__del__(self)
#学生类
class Student(SchoolMember):
    def __init__(self,name,mark):
        SchoolMember.__init__(self,name)
        self.mark = mark
```

```
    def say_hello(self):
        SchoolMember.say_hello(self)
        print("我是学生,我的成绩是%d" % self.mark)
    def __del__(self):
        SchoolMember.__del__(self)

t=Teacher("Andrea","1502")
t.say_hello()
s=Student("Cindy",77)
s.say_hello()
```

程序运行结果：

```
学校新加入一个成员：Andrea
学校共有 1人
大家好,我叫 Andrea
我是老师,我带的班级是 1502班
学校新加入一个成员：Cindy
学校共有 2人
大家好,我叫 Cindy
我是学生,我的成绩是 77
Andrea离开了,学校还有 1人
Cindy离开了,学校还有 0人
```

本例程序中定义了一个父类 SchoolMember 类，类成员包括 member 和 name 属性与 say_hello()方法。在对象创建时，调用__init__()构造函数；在对象使用结束时，调用__del__()函数。Teacher 类与 Student 类继承于 SchoolMember 类，可直接使用父类的 member、name 公有属性和 say_hello()公有方法。另外，Teacher 类定义子类属性 grade，Student 类定义子类属性 mark。

习　题

1. 设计一个类 Flower，创建两个对象属性与两个方法，建立 Flower 的两个实例化对象并使用。

2. 创建一个 Fruit 类为基类，包括若干属性及方法，两个子类 Apple、Pear 分别继承于 People 类，并在子类中分别创建新的属性与方法，实例化对象并使用。

第 12 章 图形用户界面设计

12.1 图形用户界面的选择与安装

图形用户界面(graphical user interface,GUI)是用户和程序交互的媒介,向用户提供了一种图形化的人机交互方式,它为程序提供一组界面组件,GUI 应用程序向用户呈现出一套直观、新颖又极易使用的操作界面,使人机交互变得更简单、更直接。应用程序越复杂,对 GUI 的要求越高。GUI 有助于用户掌握新程序,降低使用开销,提高用户程序使用效率。

常用的 GUI 工具有 Jython、IronPython、Tkinter、wxPython 几种。其中,Jython 针对 Java 语言提出;IronPython 针对.NET 用户,支持标准的 Python 模块;Tkinter 是针对 Python 语言的 GUI 库;wxPython 针对 C++ 用户。下面简单介绍 Tkinter 和 wxPython 的安装。

1. 安装 Tkinter

Tkinter 是一款流行的跨平台 GUI 工具包,是 Python 标准的 GUI 库,Python 自带的 IDLE 就是用它编写的。Tkinter 是 Tk GUI 系统的 Python 界面。Tkinter 是 Python 自带的,不需要安装,是 Python 创建 GUI 最常用的工具。它只提供了非常基本的功能,而未提供工具栏、可停靠式窗口及状态栏,但可以通过其他方法创建这些控件。在 Windows(64/32)/Linux/UNIX/Macintosh 操作系统下均能使用。下载地址为 http://www.python.org/topics/tkinter。下载完成后,结合网站提供的演示程序和开发文档进行安装配置后即可使用。

2. 安装 wxPython

wxPython 非默认包,需自行下载安装稳定版本 wxPython 2.8。下载地址为 http://wxpython.org/download.php。下载后双击文件,按默认参数安装后即可使用。

12.2 图形用户界面程序设计基本问题

GUI 程序的基础是其根窗体(root window),各 GUI 元素均放在窗体上,若将 GUI 视为一棵树,根窗体就是树根,树的分支均来自树根。我们需先引入 tkinter 模块,再实例化 tkinter 模块的 Tk 类,如 root=Tk()。这里不需要在类名 Tk 前面加上模块名 tkinter,直接访问 tkinter 模块中的任何部分,可以创建一个根窗体,而每个 Tkinter 程序只能有一个根窗体,可通过这个根窗体的一些方法对其进行修整。

GUI 元素被称为控件，Label 是较小的控件，它是一种很重要的控件，表示的是不可编辑的文本或图标，用于标记 GUI 中的各部分和其他控件。不同于其他诸多控件，Label 具有不可交互的特性。Tkinter 工具包中有 tkinter 模块，通过实例化该模块中的类的对象，就能创建出 GUI 元素。表 12.1 给出部分 GUI 核心窗口控件（widget）说明和其对应的 tkinter 类。

表 12.1　部分 GUI 核心窗口控件说明和其对应的 tkinter 类

控件及类	说　　明	控件及类	说　　明
Frame	承载其他 GUI 元素	Radiobutton	允许用户从多个选项中选取一个
Label	显示不可编辑的文本或图标	Menu	与顶层窗口相关的选项
Button	用户激活按钮时执行一个动作	Scrollbar	滚动其他控件的滚动条
Entry	接受并显示一行文本	Canvas	图形绘图区：直线、圆、照片、文字等
Text	接受并显示多行文本	Dialog	通用对话框的标记
Checkbutton	允许用户选择或反选一个选项		

Tk 到 tkinter 的映射如表 12.2 所示。

表 12.2　Tk 到 tkinter 的映射

操　作	Tcl/Tk	Python/tkinter
Creation	frame. panel	panel=Frame()
Masters	button. panel. quit	quit=Button(panel)
Options	Button. panel. go-fg-black	go=Button(panel,fg='black')
Configure	. panel. go config-bg red	Go. config(bg='red') go['bg']='red'
Actions	. popup invoke	Popup. invoke()
Packing	Pack. panel-side left-fill x	Panel. pack(side=LEFT,fill=X)

【例 12.1】　“Hello World!”程序窗口。

程序如下：

```
#coding=GBK
from tkinter import *
root=Tk()
Label(root,text='Hello World!').pack()
root.mainloop()
```

程序运行结果如图 12.1 所示。

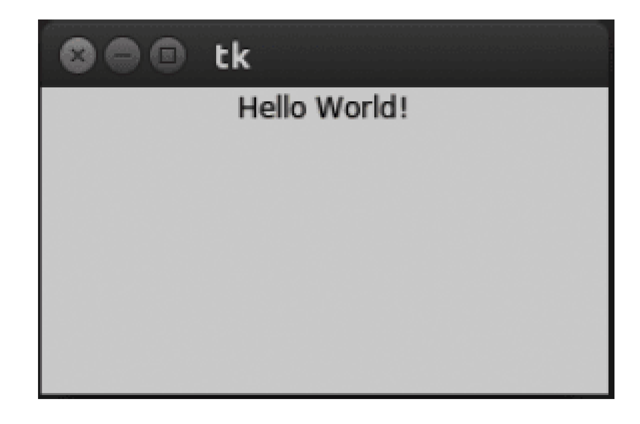
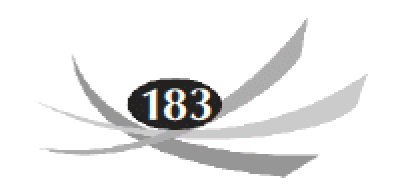


图 12.1　输出“Hello World!”

框架（Frame）是一个用来承载其他控件（如 Label）的控件，通常称为窗体。通常一个框架包含一个单一的窗体，更多子窗体被放置在这个窗体中，框架的唯一子窗体的尺寸自动随其父框架尺寸的改变而改变。在创建控件的时候，



必须将其容器(master,承载这个控件的东西)传给它的构造器,新框架就被放到根窗体里,而 grid()是所有控件都有的方法,它关联了一个布局控制器,来安排控件的布局。

通过实例化 Label 类的一个对象,创建出一个 Label 控件,如 lbl=Label(root,text="I'm a label!"),将 root 传给 Label 对象的构造器,则 root 所引用的那个框架就成了这个 Label 控件的容器。可通过设置控件可供设置的选项来设置控件的外观,通过调用 lbl 对象的 grid()方法,如 lbl.grid(),确保该标签是可见的。

通常,一个应用程序包含导入必需的包、建立框架类、建立主程序三个基本步骤,而主程序通常满足建立应用程序对象、建立框架类对象、显示框架、建立事件循环四个功能,通过调用根窗体的事件循环以启动 GUI,如调用 root.mainloop()就会打开窗口并等待处理将要发生的事件。而应用程序的实现基于应用程序对象和顶级窗体两个必要对象,任何应用程序都需要实例化一个对象并至少有一个顶级窗体。应用程序至少有一个 Frame 的子类,Frame 对象可以通过 style 参数来创建组合样式,每个 Frame 对象都有一个 ID,ID 由应用程序生成或由应用程序显式赋值。

大多数 GUI 应用程序都遵循标准的创建流程:先创建用于表示程序中窗口的一个类或者多个类,其中之一为主窗口;再针对每个窗口类,创建此窗口需要用到的变量,创建控件,调整控件布局并指定一些方法用于响应各种事件。GUI 应用程序分为对话框式程序和主窗口式程序。前者是既没有菜单又没有工具栏的窗口,用户一般通过窗口中的按钮及下拉列表框等控件与之交互;后者则有中心区域,区域上方通常有菜单与工具栏,下方有状态栏,而且还可能带有可停靠式窗口。

12.3　常用控件

12.3.1　按钮

1. 创建按钮

按钮(Button)控件可以通过用户激活而执行某个动作,通过实例化 Button 类的一个对象,创建出一个 Button 控件。例如:

```
#在框架中创建一个按钮
bttn1=Button(app,text="yes!")
bttn1.grid()
```

此按钮的容器是之前创建的框架,即这个按钮被放置在那个框架上。而创建控件后,可以使用该对象的 configure()方法对控件的任何选项进行设置,还可以用 configure()方法修改已经设置好的相关选项。

2. 命令按钮

一般使用命令按钮响应用户的鼠标单击操作。在 wxPython 中将控件直接放置在框架上,默认不能让具体的窗体内容和其他工具栏、状态栏分开。为了将其分开,在框架上放置控件时,以 Tab 键遍历窗体实例中的元素,一般会创建与框架大小相同的窗体用来容纳框架中的全部内容。

如先在框架中放置窗体,在其上放置“关闭”按钮 Close,在程序运行时单击该按钮,窗体关闭且应用程序退出。其中,窗体可以不用定义位置和大小。在 wxPython 中,若仅一个子窗体的框架被创建,窗体将会自动调整大小填满该框架的客户区域;Close 按钮是窗体的元素,默认在窗体右上角,默认大小是按钮标签长度,具体大小和位置需被指定;子窗体按钮的大小不随窗体大小的变化而变化,但实现较复杂的布局可以通过 Size 对象来管理子窗体。

12.3.2 文本控件

在 GUI 编程中,通常需要让用户输入一些文本或向用户显示一些文本,可以通过文本控件来实现。这样就可以通过读取文本控件中的内容或在文本控件中插入文本以向用户提供相应的反馈信息。本节主要以 wxPython 包中的方法为例介绍文本控件。

1. 静态文本框

不影响鼠标操作,用户不能更改显示的文本称为静态文本。一般使用静态文本框显示提示性信息。在屏幕上只显示纯文本是所有可视化用户界面最基本的任务。wxPython 中用 wx.StaticText 类实现静态文本,它能设置文本对齐方式、字体和颜色,多行文本通过带换行符\n 的字符串实现。wx.StaticText 默认从 wx.Window 父类继承方法,其构造函数格式如下:

```
wx.StaticText(parent,id,label,pos,size,style,name)
```

其中,parent 是父窗体控件;id 是标识符,具有唯一性;label 是欲显示在控件中的文本;pos 是一个 Python 元组,是窗体控件的位置;size 是 wx.size,是窗体控件的大小;style 是样式标记;name 是对象的名字,帮助用户查找对象。表 12.3 给出专用于 wx.StaticText 的样式。

表 12.3 wx.StaticText 专用样式

样 式	说 明
wx.ALIGN_CENTER	文本在静态文本框的中心
wx.ALIGN_LEFT	文本在静态文本框中左对齐,默认样式
wx.ALIGN_RIGHT	文本在静态文本框中右对齐

wxPython 的默认大小恰好是包容文本的大小,所以当创建一个非默认样式的单行静态文本时,需要显式设置控件大小。

2. 文本框

在可视化编程中,程序需要用文本框来接收用户从键盘输入的信息,一般通过它来接收用户的输入和显示计算结果。而 wx.TextCtrl 类用于 wxPython 文本域窗体控件,分为单行文本框和多行文本框,常用于输出信息,也能作密码输入控件。wx.TextCtrl 类的构造函数格式如下:

```
wx.TextCtrl(parent,id,value,pos,size,style,validator,name)
```



其中,参数 parenr、id、pos、size、style 和 name 与 wx.StaticText 构造函数的参数相同;value 是显示在该控件中的初始文本;validator 用于数据过滤以保证只能输入要接受的数据。

单行文本控件的专用样式如表 12.4 所示,类似于其他样式标记,它们能通过“|”符号组合使用。

表 12.4　单行文本控件专用样式

样　　式	说　　明
wx.TE_CENTER	控件中文本居中
wx.TE_LEFT	控件中文本左对齐,默认样式
wx.TE_RIGHT	控件中文本右对齐
wx.TE_NOHIDESEL	文本始终高亮显示,仅适用 Windows 系统
wx.TE_PASSWORD	隐藏输入文本,以星号显示
wx.TE_PROCESS_ENTER	使用此样式,用户在控件内按回车键时,触发一个文本输入事件
wx.TE_READONLY	文本控件为只读,其中文本不能修改

在 Tkinter 中,会用单行文本框(Etry)和多行文本框(Text)来接收用户从键盘输入的信息。Text 中参数 width 和 height 设置多行文本框的尺寸,参数 wrap 决定文本换行方式,取值有 WORD、CHAR、NONE 等。值为 WORD 时,遇到文本框的右边缘,整个单词自动换行;值为 CHAR 时,遇到文本框的右边缘,只将下一个字符放到下一行;值为 NONE 时,不能自动换行,即只能在文本框的第一行写字。

12.3.3　菜单栏、工具栏、状态栏

框架中有明确的关于管理菜单栏、工具栏和状态栏的机制。一般通过使用菜单栏和工具栏实现复杂的程序,通过状态栏显示系统的一些提示信息。

【例 12.2】　菜单栏、工具栏和状态栏的添加和事件响应。

程序如下:

```
import wx
class Frame7(wx.Frame):
def __init__(self,superior):
    wx.Frame.__init__(self,superior,-1,'Menubars',size=(400,300))
    panel=wx.Panel(self)
    #创建状态栏,它是 wx.StatusBar 类的实例
    self.statusBar=self.CreateStatusBar()
    #创建工具栏,它是 wx.ToolBar 类的实例
    toolbar=self.CreateToolBar()
    #将工具添至工具栏,使用的是 AddSimpleTool()
    toolbar.AddSimpleTool(11,wx.Image('open.png',
    wx.BITMAP_TYPE_PNG).ConvertToBitmap(),"Open",
```

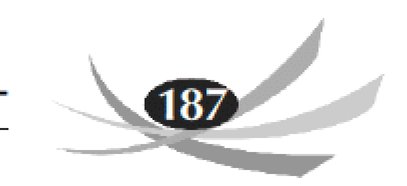
```
            "Click it to Open a file.")
        #准备显示工具栏,Realize()方法告诉工具栏这些工具的位置
        toolbar.Realize()
        #为 open 工具栏添加一个事件处理函数,响应选择某工具栏或菜单等事件
        #提供将事件处理函数绑定到程序主对象 self
        #与之相配的工具栏的 id事件处理方法的名称
        wx.EVT_TOOL(self,11,self.OnToolOpen)
        menuBar=wx.MenuBar()                #创建菜单栏
        menu1=wx.Menu()                     #创建名为 menu1 的菜单栏
        #在 menu1 下添加几个子菜单、分隔条
        #参数分别为 id,选项文本,鼠标位于其上时显示在状态栏上的文本
        menu1.Append(101,"&New","Create a New File")
        menu1.Append(102,"&Open"," ")
        menu1.Append(103,"&Close"," ")
        menu1.AppendSeparator()
        menu1.Append(104,"Close All","Close All Opened File")
        menu1.Append(105,"Exit"," ")
        menuBar.Append(menu1,"&File")       #将 menu1 添至菜单栏显示为 File

        menu2=wx.Menu()                     #创建名为 menu2 的菜单
        menuBar.Append(menu2,"&Edit")       #将 menu1 添加到菜单栏上
        self.SetMenuBar(menuBar)            #在框架上附上菜单栏
        #为 Exit 菜单项添加一个事件处理函数
        wx.EVT_MENU(self,105,self.OnMenuExit)
        #响应工具栏的操作,需要方法定义于其中的那个对象和产生的事件两个参数
        def OnToolOpen(self,event):
            self.statusBar.SetStatusText('You open a file!')
        def OnMenuExit(self,event):         #响应菜单栏 Exit 的操作
            self.Close(True)
        def OnCloseMe(self,event):
            self.Close(True)
        def OnCloseWindow(self,event):
            self.Destory()
    if __name__=='__main__':
        app=wx.App()
        frame=Frame7(None)
        frame.Show()
        app.MainLoop()
```

程序运行结果如图 12.2 所示。

12.3.4 对话框

一般来说,用户在进行设置时,会弹出一些信息对话框,让用户进行一些操作,如弹出警告对话框,需要单击 OK 按钮。wxPython 支持消息对话框、文本输入对话框、单选对



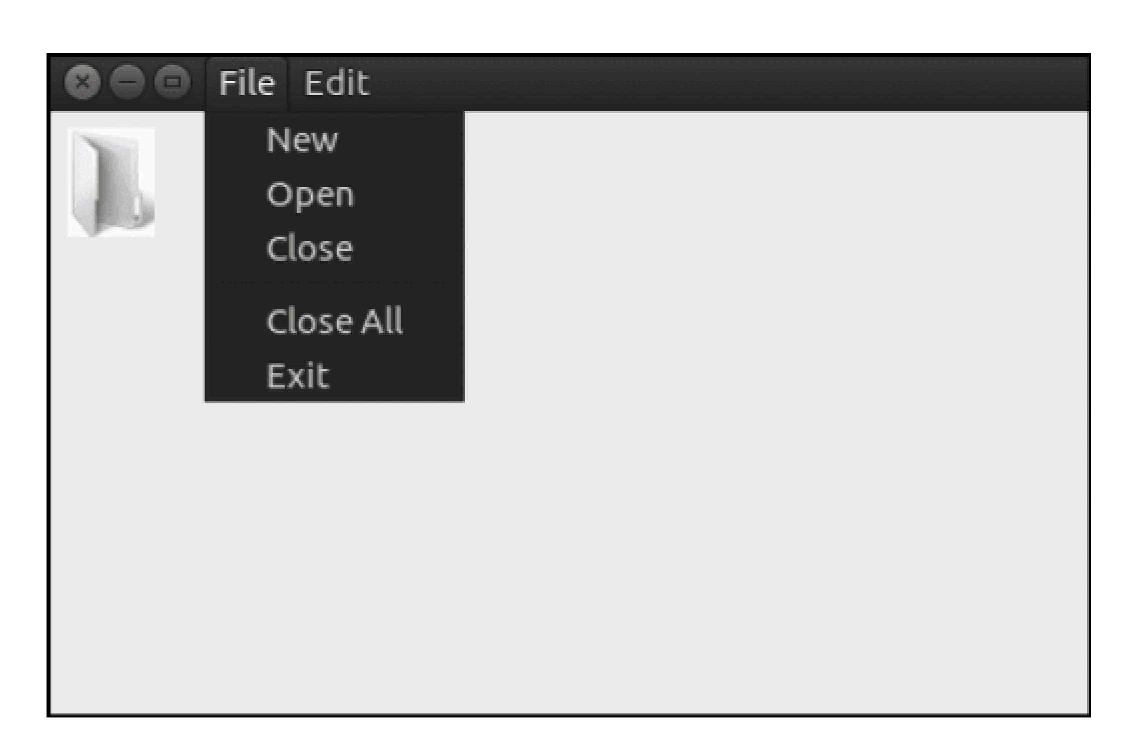


图 12.2　打开 File 菜单

话框、文件选择器、进度对话框、打印设置和字体选择器等多种预定义对话框。当想简单、快速地得到来自用户的信息时，可以为用户显示一个标准的对话框体。很多任务都有标准的对话框，包括警告框、简单的文本输入框和列表选择等。

Tkinter 在 tkinter. messagebox 中有全套的对话框，通过 tkinter. messagebox 里面的 showinfo、showwarning、showerror 函数来弹出一个警告，它们的功能基本类似，只是对话框符号不同。而其他对话框可以在颜色选择器 tkinter. colorchooser 和文件选择器 tkinter. filedialog 包里找到，这里不再赘述。

对话框按模态分为如下四种。

- 全局模态(global modal)：它会阻塞整个操作系统的用户界面，使用户只能与本对话框交互，而不能操作其他应用程序。有两种用法：一个是作为操作系统启动时的登录对话框；另一个是从上了密码的屏幕保护程序中跳出时所显示的解锁框。
- 应用程序级模态(application modal)：它会阻止用户操作程序里的其他对话框，但用户仍然可以切换至系统里的其他应用程序。
- 窗口级模态(window modal)：类似于应用程序级模态，但它并不能完全阻止用户操作应用程序里的其他对话框，而是只能组织用户操作位于同一对话框体系里的其他对话框。
- 非模态/无模态(modeless)：非模态对话框既不会阻塞本应用程序中的对话框，也不会阻塞其他应用程序中的对话框。它编写起来比模态对话框困难。

对于控件、布局及事件绑定，模态对话框和非模态对话框无区别。前者会把用户输入的内容赋值给相关变量；后者则通常会调用应用程序的方法或修改应用程序数据，以便响应硬化操作。

1. 消息对话框

简单的消息提示框 wx. MessageDialog 参数如下：

```
wx.MessageDialog(parent,message,caption,style,pos)
```

wx. MessageDialog 返回值为常量 wx. ID_YES、wx. ID_NO、wx. ID_CANCEL 和 wx. ID_OK 之一。其中，parent 是对话框父窗体，顶级为 None；message 是显示在对话框

中的提示信息，为一系列字符串；caption 是对话框标题栏上显示的字符串；style 是对话框中的按钮样式，最常用的 wx. YES_NO 只有“是”和“否”两个按钮，wx. CANCEL 表示“取消”，wx. OK 表示“确定”，wx. ICON_QUESTION 会在提示信息前加一个“?”图标；pos 是对话框位置。

在定义消息对话框时会经常用到 ShowModal()方法，它以模式框架的方式显示对话框，也就是在对话框关闭之前，应用程序中其他窗体不能响应用户事件，其返回值为整数。

2. 文本输入对话框

获取用户输入的一行文本时，可以用文本框，也可以用文本输入对话框 wx. TextEntryDialog，其参数为：父窗体、显示在窗体中的提示信息（文本标签）、窗体标题，窗体标题默认值是 Please enter text，输入框中的默认值、样式参数默认为 wx. OK | wx. CANCEL。

与消息对话框相同，文本输入对话框的 ShowModal()方法返回单击按钮的 ID，可以通过 GetValue()方法获取文本框中的输入值。

3. 单选对话框

wx. SingleChoiceDialog 类能让用户在提供的列表中选择，其参数与文本输入对话框类似，通过字符串列表代替默认字符串文本。用户要想获取刚才的列表选择结果可以使用 GetSelection()方法和 GetStringSelection()方法，前者返回用户选项索引，后者返回用户选项的字符串。

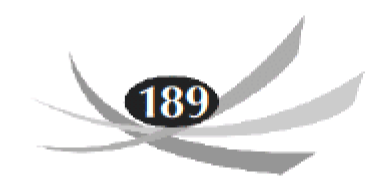
12.3.5 复选框

复选框是带有文本标签的开关按钮，它允许用户从一组选项中选取任意数量的选项。存在多个复选框时，各复选框的开关状态独立。任何复选框都要有一个特殊的对象与之关联，用于自动反应该复选框的状态。复选框给 GUI 编程提供极大的灵活性，也让用户更好地对程序进行控制。复选框的构造函数如下：

```
wx.CheckBox(parent,id=-1,label=wx.EmptyString,pos=wx.DefaultPosition,
size=wx.DefaultSize,style=0,name="checkBox")
```

label 参数是复选框标签文本。EVT_CHECKBOX 是复选框常用事件，单击复选框时触发该事件。常用的方法是：引用布尔对象的复选框状态方法 GetValue()得到布尔值，选中复选框时返回 True，否则返回 False；SetValue(True)表示选中复选框，SetValue(False)表示取消复选框的选中状态。若用户选中或者清除这个复选框时，这些方法所在的 update_text()方法就会被调用。

Tkinter 采用一种很特殊的方式来访问界面控件中的内容。当需要获取或设置文本框、标签等控件内容时，可以创建一些特殊对象。这些对象有很多种形式，其中 String Var 最常用。而在复选框使用中，若想在对话框上放置一个复选框，此时需要通过一个特殊“变量”String Var 来确定复选框的选择状态，但如果取值为数字，则特殊“变量”类型为 IntVar。



12.3.6　单选框

单选框类似于复选框，向用户提供两种或两种以上的选项，但只允许用户在一组选项中最多只能有一个被选择，因此不需像复选框那样为每个单选框都加上单独的状态变量。一组单选框只需共享一个用于说明“哪个单选框被选中”的特殊对象即可，它与复选框的区别是，选择新的选项后，上次的选项会取消。单选框的构造函数如下：

```
wx.RadioButton(parent,id=-1,label=wx.EmptyString,pos=wx.DefaultPosition,
size=wx.DefaultSize,style=0,validator=wx.DefaultValidator,name= "radioButton")
```

根据用户需求要分组使用单选框时，可以用 wx. SashWindow 控件的对象进行分组，每组单选框的 parent 与对象名保持一致。除此之外，控件的对象也能用于分组，或通过样式分组，每组第一个元素使用 wx. RB_GROUP 样式，其他元素均不使用该样式。EVT_RADIOBOX 是单选框常用事件，单击单选框时触发该事件；常用方法及功能与复选框一致。

12.3.7　列表框

为了展示列表中的内容，列表框使列表中的一个或者多个项可以被选中，它向用户提供多个元素(均为字符串)，便于用户选择。列表框的构造函数如下：

```
wx.ListBox(parent,id,pos=wx.DefaultPosition,size=wx.DefaultSize,
choices=None,style=0,validator=wx.DefaultValidator,name="ListBox")
```

selectmode 属性控制列表框中的选项，这样可将其设置成如下其中的一种。

- SINGLE：一次只选一个。
- BROWSE：与 SINGLE 类似，但是只允许使用鼠标选择。
- MULTIPLE：按住 Shift 键，然后用鼠标左键可以选择多行。
- EXTENDED：与 MULTIPLE 类似，但是可以使用 Ctrl＋Shift 键同时单击选择范围。该属性设置必须使用 curselection 方法找出列表框中选择的项，不需要输入数据，然后输出。

列表框的多个样式可通过运算符“|”连接，其常用样式如表 12.5 所示，常用方法如表 12.6 所示。

表 12.5　列表框常用样式

样式名称	说　明
wx. LB_EXTENDED	用户通过 Shift 键和鼠标选择连续元素
wx. LB_MULTIPLE	支持多选且选项可以不连续
wx. LB_SINGLE	仅支持单选，最多选择一个元素
wx. LB_ALWAYS_SB	列表框始终显示一个垂直滚动条

续表

样式名称	说　明
wx.LB_HSCROLL	列表只在需要时显示一个垂直滚动条,默认样式
wx.LB_SORT	使列表框中的元素按字母顺序排列

表 12.6　列表框常用方法

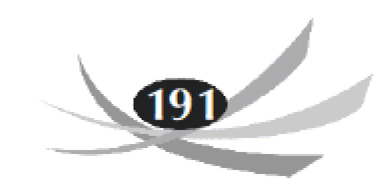
方　法	功能及说明	举　例
Append()	在列表框尾部添加一个元素	lstBox.Append(s)
Clear()	删除列表框中的所有元素	lstBox.Clear()
Delete()	删除列表中索引为 n 的元素,列表中的元素索引从 0 开始	lstBox.Delete(n)
FindString()	返回元素索引,若未找到元素则返回−1	i=lstBox.FindString('Friday')
GetCount()	返回列表中元素个数	n=lstBox.GetCount()
GetSelection()	返回当前选择项的索引,仅对单选列表有效	i=lstBox.GetSelection()
SetSelection()	用布尔值更改索引为 n 的元素的选择状态	lstBox.SetSelection(n,select)
GetStringSelection()	返回当前选择的元素,仅对单选列表有效	s=lstBox.GetStringSelection()
GetString()	获取索引为 n 的元素	s=lstBox.GetString(i)
SetString(n,string)	将索引为 n 的元素设为 s	lstBox.SetString(n,s)
GetSelection()	返回包含所选元素索引的元组	t=lstBox.GetSelection()
IsSelected()	在列表中 pos 位置前插入列表中的字符串	lstBox.IsSelected(n)
InsertItems()	返回索引为 n 的元素的选择状态的布尔值	lstBox.InsertItems(items,pos)
Set()	用 choices 的内容重新设置列表框	lstBox.Set(choices)

列表框有 EVT_LISTBOX 和 EVT_LISTBOX_DCLICK 两个常用事件,前者在列表中一个元素被选择时触发,后者在列表被双击时触发。

12.3.8　组合框

组合框继承了文本框和列表框的特点与方法,由文本框和列表框组成,适用于单选框的方法几乎均适用于组合框。其构造函数如下:

```
wx.ComboBox(parent,id=-1,value="",pos=wx.DefaultPosition,
size=wx.DefaultSize,choices=[],style=0,validator=wx.DefaultValidator,name="comboBox")
```



12.4　对象的布局

当向一个框架中放置一些控件后，就需要一种对它们进行合理组织的手段，将控件指定到对应位置，控制 GUI 的外观，而布局管理器就是这样的一种手段。时下流行的大多数 GUI 工具包均使用布局(layout)来排布控件，而不是将控件的大小及位置写成固定值，通过这种方式，每个控件既可以保持与其他控件的相对位置关系，同时又能自动扩大或缩小尺寸，以便与其内容相吻合。Tkinter 提供 grid、pack 和 place 三种完全不同的布局管理器，所有 Tkinter 控件都包含专用的布局管理方法，用来组织和管理整个父控件区中子控件的布局。

12.4.1　grid 布局管理器

网格(grid)控件对象的 grid()方法布局通用格式为：WidgetObject.grid(option=value,…)，它按网格组织控件，将控件按行、列位置放置在网格里。此布局最为流行，用起来也最为简单。宿主控件将内部空间按行和列分成若干单元格，每个单元格内可放置一个控件。grid()用行、列确定位置，列宽由列中最宽单元格确定，行高由行中最高单元格决定，行列交叉处为一个单元格，创建的单元格必须相邻。若用户进行某种操作时需要控件跨越多个单元格，连接若干单元格为一个更大空间，称此操作为跨越。组件并非填充整个单元格。用户能分配使用单元格中的剩余空间，这些空间可以空置，也可以在水平、竖直或两个方向上填充这些空间，灵活易用。用它设计对话框和带有滚动条的窗体效果极佳。其常用参数和函数如表 12.7 和表 12.8 所示。

表 12.7　grid()方法常用参数

参数名称	说　　明	取 值 范 围
row	设置控件中单元格行数	自然数，默认值从 0 开始累加
rowspan	设置单元格纵向跨越的行数	自然数，默认值从 0 开始累加
column	设置控件中单元格列号	自然数，默认值从 0 开始累加
columnspan	设置控件中单元格横向跨越的列数	自然数，起始默认值为 0
ipadx ipady	设置控件内部 x(y)方向上的空间大小；默认单位为像素，可选单位为厘米(cm)或毫米(mm)；i(英寸，in)、p(打印机的点，即 1/27in)；使用时需在值后加以上任一个后缀	非负浮点数，默认值为 0.0
padx pady	设置控件周围 x(y)方向上的空间大小；默认单位为像素，可选单位为厘米(cm)或毫米(mm)；i(英寸，in)、p(打印机的点，即 1/27in)；使用时需在值后加以上任一个后缀	非负浮点数，默认值为 0.0
in_	将该控件作为所选组建对象的子控件，即重新设置 w 为窗体 w2 的子窗体	已经 pack 后的控件对象
sticky	设置控件对齐方式，默认为“center”	“n”(北)，“s”“w”“e”“nw”“sw”“se”“ne”“center”

通过 row 和 column 参数可以定义对象在容器中的具体位置，若二者的值都为 0，则这个对象就会被放置于框架的左上角。columnspan 参数用于横跨多列放置控件，也可以用 rowspan 参数跨越多行放置小部件。在确定了控件所占单元格后，可以利用参数 sticky 调整控件在这个单元格内部的位置，它以方位为值，控件会根据方位信息移动到单元格对应位置。如在下面创建一个左对齐的标签：

```
#创建表示密码的标签
self.pw_lbl=Label(self,text="Password:")
self.pw_lbl.grid(row=1,column=0,sticky=w)
```

表 12.8　grid()方法常用函数

函数名	说　　明
slaves()	以列表方式返回该控件的所有子控件对象
propagate(boolean)	设置为 True 指父控件的几何大小由子控件决定(默认值)，反之则无关
info()	返回 pack 提供的选项的对应值
forget()	unpack 控件，将控件隐藏并忽略原有设置时，对象依旧存在，用 pack(option=value，…)能将其显示
grid_remove ()	无

12.4.2　pack 布局管理器

pack(填充)布局管理器根据某个假想的中心点来排布控件。对于比较简单的对话框来说，采用 pack()布局比较合适。pack()布局的通用格式为：WidgetObject. pack(option=value，…)，它采用块方式组织控件，将所有控件组织为一行或一列，用户能使用参数控制控件样式。pack()管理程序时以控件创建的顺序将控件添加到父控件中，通过设置相同的锚点能将一组控件依次放置在同一个位置，默认在父窗体中自顶向下添加控件。若是几个控件的简单布局，用 pack()布局的代码量最少，因此被广泛用于快速生成的界面设计中。其常用参数和函数如表 12.9 和表 12.10 所示。从表 12.9 中可以看出，expand、fill 和 side 会相互影响。

表 12.9　pack()方法常用参数

参数	说　　明	取 值 范 围
expand	控件居中，当值为“yes”时，side 选项无效；若 fill 选项为“both”，则填充父控件剩余空间	(“yes”，自然数)或(“no”，0)，默认值为“no”或 0
fill	设置 x(y)方向上的空间，当属性 side=“top”或“bottom”时，设置 x 方向；当属性 side=“left”或“right”时，设置 y 方向；当 expand 选项为“yes”时，设置父控件剩余空间	取值为“x”“y”“both”，默认值为待选
ipadx ipady	设置控件里面 x(y)方向上的空间大小；默认单位为像素，可选单位为厘米(cm)或毫米(mm)；i(英寸)、p(打印机的点，即 1/27in)；使用时在值后加以上一个后缀	非负浮点数，默认值为 0.0

续表

参数	说　　明	取 值 范 围
padx pady	设置控件周围 x(y)方向上的空间大小;默认单位为像素,可选单位为厘米(cm)或毫米(mm);i(英寸)、p(打印机的点,即 1/27in);使用时在值后加以上一个后缀	非负浮点数,默认值为 0.0
side	设置在父控件中的位置	“top”(默认),“bottom”“left”“right”
before	先创建该组件再创建选定控件	已经填充后的控件对象
after	先创建选定组件再创建该控件	已经填充后的控件对象
in_	将该控件作为所选组建对象的子组件	已经填充后的控件对象
anchor	设置对齐方式:左对齐“w”,右对齐“e”,顶对齐“n”,底对齐“s”	“n”“s”“w”“e”“nw”,“sw”“se”“ne”“center”(默认)

表 12.10　pack()方法常用函数

函　数　名	说　　明
slaves()	以列表方式返回该控件的所有子控件对象
propagate(boolean)	设为 True 指父控件的几何大小由子控件决定(默认值),反之则无关
info()	返回 pack 提供的选项所对应的值
forget()	unpack 组件,将控件隐藏且忽略原有设置而对象依旧存在,用 pack(option=value,…)能将其显示
location(x, y)	x, y 是以像素为单位的点,返回值表示此点是否在单元格中,若在单元格中则返回单元格行、列坐标,(-1, -1)表示不在单元格中
size()	返回控件中的单元格,指示控件大小

12.4.3　place 布局管理器

place(放置)布局管理器用固定值来描述控件位置,这种布局很少使用。它对全部基础控件均可用,通过它显式设置控件的大小和位置。

一般不用 place 布局管理器,因此这里不做详述。但会在特殊情况下用 place 布局管理器,如能通过 place 布局管理器将子控件显示在父控件的正中央。推荐使用 grid 布局管理器,而且 pack 和 grid 同时使用可能会导致程序崩溃。

12.4.4　布局管理器举例

【例 12.3】　采用布局管理器创建一个窗口,它左边有一个列表,其尺寸不变,右边的信息显示区会根据内容调整窗口尺寸。

程序如下:

```
from Tkinter import *


class App:
```

```
    def __init__(self,master):
        frame = Frame(master)
        frame.pack(fill=BOTH,expand=1)
                                                    # listbox
        listbox = Listbox(frame)
        for item in ['red','green','blue','yellow','pink',]:
            listbox.insert(END,item)
        listbox.grid(row=0,column=0,sticky=W+E+N+S)
                                                    # Message
        text = Text(frame,relief=SUNKEN)
        text.grid(row=0,column=1,sticky=W+E+N+S)
        text.insert(END,'world' * 100)

if __name__=='__main__':
        root = Tk()
        root.geometry('500*300')
        Label(root,text='Text',font=('Arial',20)).pack()
        App(root)
        root.mainloop()
```

上述程序中，通过控件的 sticky 属性决定小单元格位置。运行结果如图 12.3 所示。

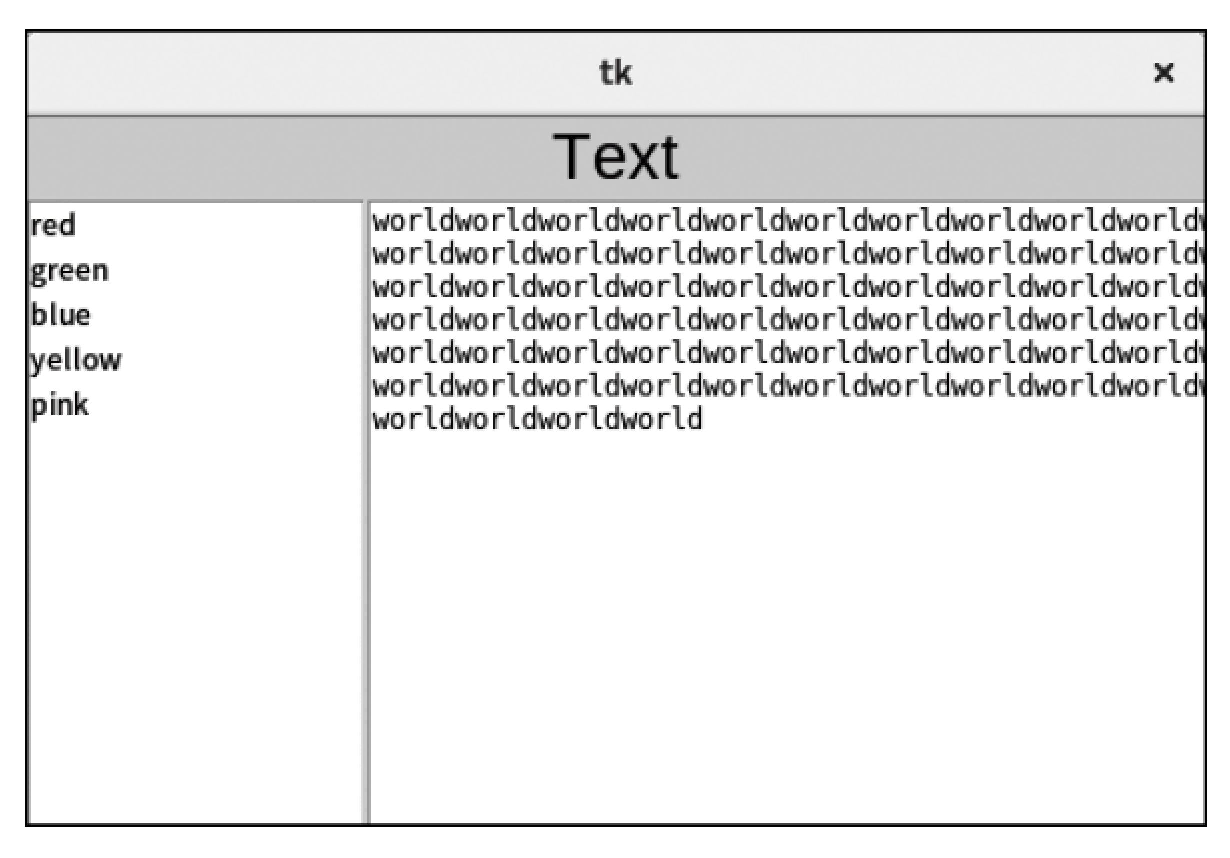


图 12.3　布局管理器界面

12.5　事件处理

事件(event)指可能会发生在对象上的事，要求有相应响应。它是一个信号，告知应用程序有重要情况发生。GUI 程序通常是由事件驱动的，编写事件驱动程序时需将事件跟程序处理器绑定起来。如最简单的用户按键盘上的某个键或单击移动鼠标。对于上述事件，程序需要做出反应，称为事件处理。Tkinter 提供的控件一般均含有诸多内在行为，如当单击按钮时执行特定操作或聚焦在一个输入栏时，用户又按了键盘上的某些键，

用户输入的内容则会在输入栏中显示，而 Tkinter 的事件处理允许用户创建、修改或删除以上行为。

12.5.1　事件处理程序

事件处理程序是当事件发生时需要执行的代码，是相应事件发生时调用的过程，大多数程序由事件驱动。用于处理单击按钮时所发生的事件的 update_count()方法如下所示：

```
def update_count(self):
    "Increase click count and display new total."
    self.bttn_clicks+=1
    self.bttn["text"] = "Total Clicks:"+ str(self.bttn_clicks)
```

该方法会统计按钮被单击的总次数，再修改按钮的文本以反映出这个新的总数。

12.5.2　事件绑定

将程序中事件发生时的被调函数称为事件处理者；当用户为自己的程序建立一个处理某个事件的事件处理者，称之为事件绑定。一般来说，设置控件的 command 选项就能将控件的动作跟一个事件处理器绑定起来，因此通常需要定义绑定事件处理器，如在 create_widget()方法中创建一个按钮：

```
def create_widget(self):
    "Create button which displays number of clicks."
    self.bttn=Button(self)
    self.bttn["text"] = "Total Clicks:0"
    self.bttn["command"]= self.update_count
    self.bttn.grid()
```

上述代码将 Button 控件的 command 选项设置为 update_count()方法，用户单击上述创建的按钮时，update_count()方法就会被调用，上述代码所做的事情就是将一个事件(Button 控件的单击事件)跟一个事件处理器(update_count()方法)绑定起来。

下面介绍绑定级别。

(1) 实例绑定：将事件和一特定控件实例绑定。如用户在处理 Canvas(画布)控件翻页时需要将按 PageUp 键事件和一个 Canvas 控件实例绑定，Tkinter 中的 Canvas 控件和窗口控件一样，可以直接添加到窗口中。调用控件实例的 bind()函数为控件实例绑定事件，若用户声明一个 Canvas 控件对象 canvas，在对象 canvas 上实现单击鼠标中键时画一条线的功能，其实现方式为：

```
canvas.bind("<Button-2>",drawline)
```

其中，<Button-2>参数是事件描述符，指定无论何时在 canvas 上单击鼠标中键时就会调用事件处理函数 drawline()执行画线任务。值得注意的是，drawline 后的圆括号是省略的，Tkinter 会将此函数填入相关参数后调用运行。

(2) 类绑定：将事件和某个控件类绑定。如用户能绑定按钮控件类，使全部按钮实例都能处理鼠标中键事件提供相应的操作。bind_class()函数能调用任何控件实例为特定控件类绑定事件。如假设用户声明了若干个 Canvas 控件对象，并想在上述对象上实现单击鼠标中键时都能画一条线的功能，其实现方式为：

```
widget.bind_class("Canvas","<Button-2>",drawline)
```

其中，widget 是任意控件对象。

(3) 程序界面绑定：在任何控件实例上触发某一事件，程序都提供相应处理。如用户可能会把 PrintScreen 键和程序中全部控件对象绑定，使整个程序界面可以处理打印屏幕事件。调用任何控件实例的 bind_all()函数为程序界面绑定事件，实现打印屏幕方式为：

```
widget.bind_all("<Key-print>",printScreen)
```

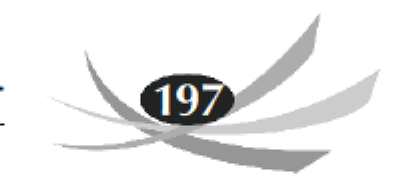
只要定义了对象、事件和事件处理器，程序的运行方式即可确定，可通过启动一个事件循环的方式来启动程序。程序在这个循环中等待已经定义好的将要发生的事，只要有事件发生，程序就会根据用户设定的方式对其进行处理。

12.6 图形用户界面设计应用举例

【例 12.4】 设计一个窗体，通过窗体上的操作来显示 Frame7。

程序如下：

```
import wx
importExp9_2_Mixbar
    class Frame11(wx.Frame):
    def __init__(self,superior):
        wx.Frame.__init__(self,superior,-1,'显示子窗体',size=(300,150))
        self.panel =wx.Panel(self)
        self.btnFrm7 =wx.Button(parent=self.panel,label='显示 Frame7',size=(100,30));
        #使用 size 控件来布置控件
        sizer=wx.FlexGridSizer(rows=2,cols=1,hgap=30,vgap=20)
        sizer.AddMany([self.btnFrm7])
        self.panel.SetSizer(sizer)
#把事件 wx.EVT_BUTTON、事件处理函数 self.OnDplFrm、按钮 self.btnFrm7 三者绑定
        self.Bind(wx.EVT_BUTTON,self.OnDplFrm7,self.btnFrm7)
    def OnDplFrm7(self,event):
        fame7=Exp9_2.Frame7(self)        #生成对象
        fame7.Show()                     #显示对象
#主程序
if __name__=='__main__':
    app=wx.App()
    frame=Frame11(None)
```




```
    frame.Show()
    app.MainLoop()
```

在上述程序中,要显示另外的窗体,首先要导入窗体所在的模块,然后可通过菜单或按钮来显示,本例中通过按钮来显示。程序运行结果如图 12.4 所示。



(a) 子窗体　　(b) 主窗体

图 12.4　子窗体与主窗体程序运行结果

【例 12.5】　设计一个图形界面的猜数字游戏。

程序如下:

```
import tkinter as tk
import sys
import random
import re

number= random.randint(0,1024)
running= True
num= 0
namx= 1024
nmin= 0

def eBtnClose(event):
    root.destroy()

def eBtnGuess(event):
     global namx
     global nmin
     global num
     global running

     if running:
        val_a = int(entry_a.get())
        if val_a == number:
            labelqval("恭喜答对了!")
            num+ = 1
```

```
            running = False
            numGuess()
        elif val_a < number:
            if val_a > nmin:
                nmin = val_a
                num+ = 1
                label_tip_min.config(label_tip_min,text= nmin)
            labelqval("太小了")
        else:
            if val_a < namx:
                namx = val_a
                num+ = 1
                label_tip_max.config(label_tip_max,text= namx)
            labelqval("太大了")
    else:
        labelqval('恭喜!答对了!')

def numGuess():
    if num == 1:
        labelqval('太厉害了,一次答对!')
    elif num < 10:
        labelqval('十次以内就答对了,尝试次数:'+ str(num))
    elif num < 50:
        labelqval('五十次以内就答对了,尝试次数'+ str(num))
    else:
        labelqval('尝试次数超过五十次了,尝试次数:'+ str(num))

def labelqval(vText):
    label_val_q.config(label_val_q,text= vText)

root = tk.Tk(className= "猜数字游戏")
root.geometry("400x90+ 200+ 200")

line_a_tip = tk.Frame(root)
label_tip_max = tk.Label(line_a_tip,text= namx)
label_tip_min = tk.Label(line_a_tip,text= nmin)
label_tip_max.pack(side = "top",fill = "x")
label_tip_min.pack(side = "bottom",fill = "x")
line_a_tip.pack(side = "left",fill = "y")

line_question = tk.Frame(root)
label_val_q = tk.Label(line_question,width= "50")
label_val_q.pack(side = "left")
line_question.pack(side = "top",fill = "x")
```




```
line_input = tk.Frame(root)
entry_a = tk.Entry(line_input,width= "40")
btnGuess = tk.Button(line_input,text= "猜 ")
entry_a.pack(side = "left")
entry_a.bind('<Return> ',eBtnGuess)
btnGuess.bind('<Button-1> ',eBtnGuess)
btnGuess.pack(side = "left")
line_input.pack(side = "top",fill = "x")

line_btn = tk.Frame(root)
btnClose = tk.Button(line_btn,text= "关闭 ")
btnClose.bind('<Button-1> ',eBtnClose)
btnClose.pack(side= "left")
line_btn.pack(side = "top")

labelqval("请输入 0～1024 任意整数：")
entry_a.focus_set()

print(number)
root.mainloop()
```

程序运行结果如图 12.5 所示。



图 12.5　猜数字游戏运行结果

习　　题

1. 将本章中的例题在 Python 中输入、调试、运行并更改例题中的一些参数，比较更改后运行效果与例题中的异同。

2. 尝试在框架中放置三个大小相同的命令按钮并在窗体中等距分布。

3. 创建一个包含四个工具的工具栏，工具分别为 New、Open、Help 和 Exit。单击前

三个工具时在窗体下方的状态栏中显示相应的提示信息，单击 Exit 工具时退出程序。

4. 计算 $1+2+3+\cdots+n$，数据输入和输出均使用文本框。

5. 创建一个名为“Order List!”的 GUI 程序，它向用户呈现一份简单的餐馆菜单，列出菜品和价钱。用户可以选取不同的菜品，然后显示总金额。

第13章 数据挖掘

13.1 关于数据挖掘

在数据大爆炸的时代，每天都有海量数据不断地进入网络。在数据时代，由于数据的爆炸式增长，人们往往被淹没在数据的海洋中却又在忍受着知识的饥渴，因此急需功能强大的工具从这些海量数据中发现有价值的信息，把数据转化成人们关注的有组织的知识。这种急切的需求导致了数据挖掘的诞生。数据挖掘是一个新兴的、面向实际应用的人工智能(artificial intelligence，AI)研究领域。1989 年 8 月，在美国底特律召开的第 11 届国际人工智能联合会议的专题讨论会上首次出现数据库中的知识发现(knowledge discovery in database，KDD)这一术语。随后，在 1991 年、1993 年和 1994 年的会议中都举行了 KDD 专题讨论会，来自各个领域的研究人员和应用开发者集中讨论数据统计、海量数据分析算法、知识表示、知识运用等问题。最初，数据挖掘是作为 KDD 中利用算法处理数据的一个步骤，其后逐渐演变成 KDD 的同义词。

数据挖掘就是从大量数据中提取出隐藏在其中的有用信息，也称数据库中的知识发现。它是从大量数据中提取出可信、新颖、有效并能被人理解的知识(模式)的处理过程，为决策提供支持，是一种深层次的数据分析方法。这里的数据往往是大量的、不完全的、有噪声的、模糊的、随机的实际应用数据。这里的知识是指人们感兴趣的概念、规则、模式、规律和约束等。这里所提取的知识，不是要去发现放之四海而皆准的真理，也不是要去发现崭新的自然科学定理和纯数学公式，更不是机器定理证明。实际上，所有提取的知识都是相对的，是有特定前提和约束条件、面向特定领域的，同时还要能够易于被用户理解。其中，模式知识是挖掘知识中最常见的，模式给出了数据特性或数据之间的关系，是对数据所包含的信息更抽象的描述。模式按功能可以分为预测型模式和描述型模式。在实际应用中，可以细分为关联模式、分类模式、聚类模式和序列模式等。

数据挖掘作为一门新兴的交叉学科，涉及数据库系统、数据仓库、统计学、机器学习、可视化、信息检索和高性能计算等诸多领域。此外，还与神经网络、模式识别、空间数据分析、图像处理、信号处理、概率论、图论和归纳逻辑等领域关系密切。

数据挖掘的过程，是一个不断调整、修改与循环的过程，其基本过程如图 13.1 所示。针对采集到的数据，数据挖掘过程可以分为三大步骤：数据预处理、数据挖掘以及评估与表示。三大步骤之间反复循环、调整，直到得到满意结果为止。

数据预处理主要包括数据清洗、数据集成、数据归约和数据变换等过程。

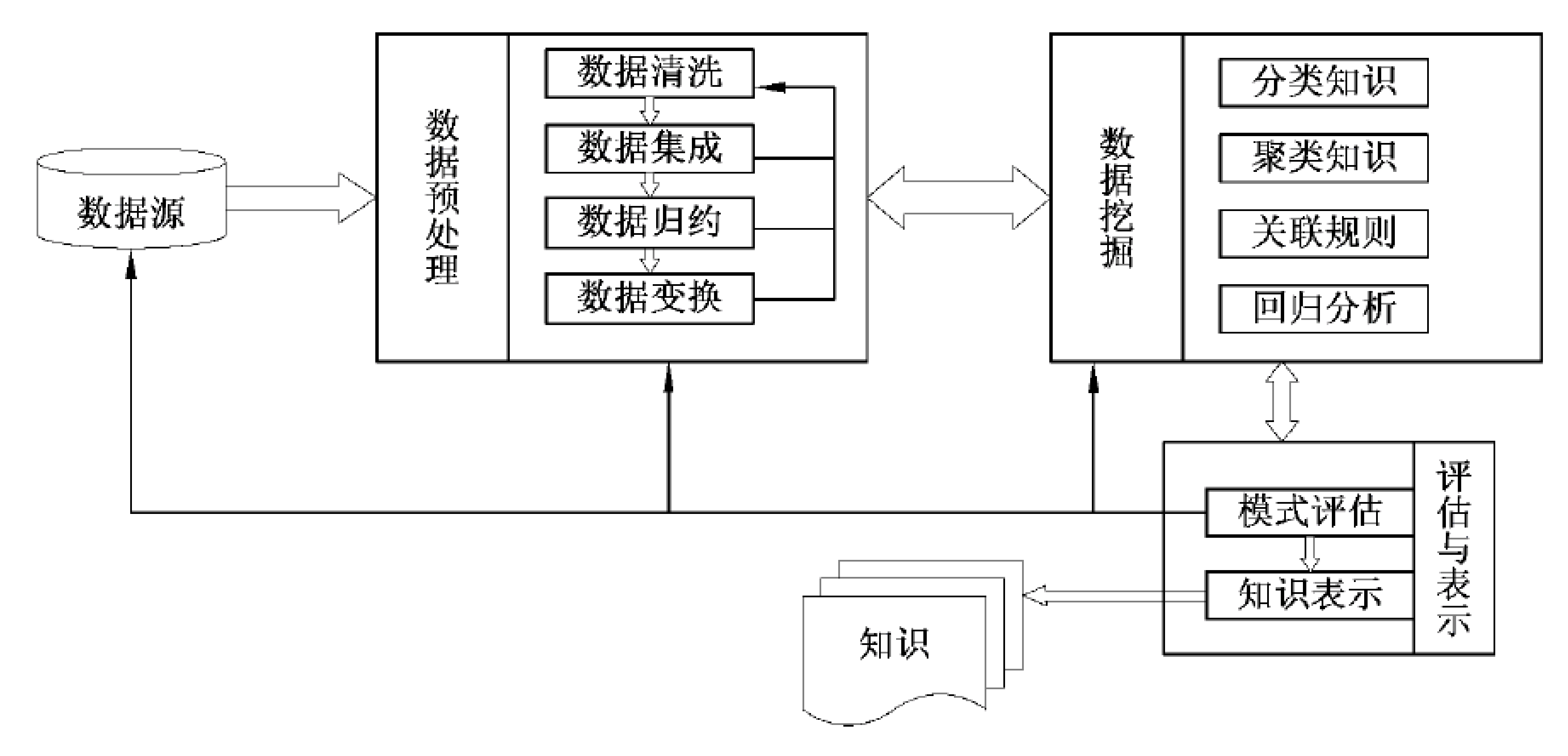


图 13.1　数据挖掘基本过程

数据清洗：因为在数据库中的数据有一些是不完整的（有些感兴趣的属性缺少属性值）、含噪声的（包含错误的属性值）、不一致的（同样的信息不同的表示方式），因此需要进行数据清洗，得到完整、正确、一致的数据。

数据集成：把不同来源、格式、特点、性质的数据在逻辑上或物理上有机地集中，从而提供全面的数据共享。

数据归约：面对海量数据进行复杂的数据分析和挖掘将需要很长时间。数据归约技术可以用来得到数据集的归约表示，数据量小很多，但仍接近原数据的完整性。常用手段包括维度归约和数量归约。维度归约就是降低数据的维数，包括特征选择和特征提取。数量归约就是利用分箱、回归、直方图、聚类等方法对数据进行离散化，减少数据量。

数据变换：通过平滑聚集、数据概化、规范化等方式将数据转换成适用于数据挖掘的形式。对于有些实数型数据，有时通过概念分层和数据的离散化来转换数据。

数据挖掘主要是通过一系列的数据挖掘算法，对经过处理的数据进行知识发现的过程；评估与表示主要包括模式评估阶段和知识表示阶段。数据挖掘的任务按照目标可以分为以下四类。

（1）分类知识：通过分析训练集的数据，为每一个分类建立分类模型，用这个已知的规律分类模型对其他数据进行分类，是一种典型的有监督学习方法。

（2）聚类知识：根据数据本身的相似性，将数据对象集合分成若干簇，是一种典型的无监督学习方法。

（3）关联规则：通过统计对象同时出现的现象，寻找给定数据集合中各个对象之间的关联关系。

（4）回归分析：确定两种或两种以上变数间相互依赖的定量关系模型，从而预测未来。

模式评估是指根据某种兴趣度量来识别表示知识的真正有趣的模式，从商业角度，由行业专家来验证数据挖掘结果的正确性；知识表示是指将数据挖掘所得到的分析信息以可视化的方式呈现给用户，或作为新的知识存放在知识库中，供其他应用程序使用。

13.2　使用 Python 进行数据挖掘

13.2.1　为什么选择 Python 进行数据挖掘

Python 是目前最受欢迎的动态编程语言之一。在解释执行语言当中，Python 以其强大而活跃的科学计算社区而著称，Python 在相关的扩展库的帮助下（如 Pandas）已经成为完成数据操控任务强有力的工具，同时 Python 相比于其他语言更加主流的优势，使得 Python 成为一个用来构建以数据为中心的应用软件的良好选择。近年来，Python 科学计算方面的功能在工业界和科学研究界中的应用显著增长。

Python 作为胶水语言，其成功有一部分归结于 Python 作为一个科学计算平台而且拥有对 C、C++、FORTRAN 代码良好的集成上，大多数现代科学计算环境都有一些遗留下来的用 C、FORTRAN 来计算一些线性代数、最优化、组合数学、快速傅里叶变换等方面的算法，很多工程师利用 Python 来黏合已经使用多年的遗留软件系统。

Python 对于一些场景也有缺陷。由于 Python 是一门解释型编程语言，因此大部分 Python 代码都要比用编译型语言（如 C、Java、C++）编写的代码运行慢很多。对于高并发、多线程的应用程序而言，Python 并不是一种理想的编程语言，因为 Python 采用全局解释器锁（global interpreter lock，GIL），每一个 iterpreter 进程只能同时仅有一个线程来执行，获得相关的锁，存取相关资源，从而防止 interpreter 同时执行多条 Python 字节码指令。

总之，Python 作为一种脚本语言和胶水语言，在大量扩展库的帮助下，非常适合数据挖掘系统的快速开发。可以将数据挖掘的框架用 Python 编写，用 Python 去实现基本想法，进行相关实验和分析，等算法成熟以后最核心的算法可以用 Java 和 C 编写，一方面把算法隐藏起来，另一方面可以提高算法的运行效率。

13.2.2　进行数据挖掘工作必要的 Python 库

Python 拥有一个巨大的活跃的科学计算社区，拥有不断改良的库（如 Numpy 和 Pandas），能够轻松地集成 C、C++、FORTRAN 代码，可以同时用于研究和原型的构建以及生产系统的构建。

1. Numpy

Numpy 是 Python 的一种开源的数值计算的基础库。它可用来存储和处理大型矩阵，比 Python 自身的嵌套列表结构（nested list structure）要高效很多。它提供了一个强大的 *N* 维数组对象 Array；比较成熟的（广播）函数库；用于整合 C/C++ 和 FORTRAN 代码的工具包；实用的线性代数、傅里叶变换和随机数生成函数。它还提供了许多高级的数值编程工具，如矩阵数据类型、矢量处理以及精密的运算库。

2. Pandas

Pandas 是基于 Numpy 的一种工具，它提供了大量的便捷处理结构化数据的数据结构和函数。它是使 Python 成为强大而高效的数据分析环境的重要因素之一。Pandas 兼具 Numpy 高性能的数组计算功能以及电子表格和关系型数据库灵活的数据处理功能，

能便捷地完成重塑、切片、切块、聚合以及选取子集等操作。

3. Matplotlib

Matplotlib 是最流行的用于绘制数据图表的 Python 库。它和 IPython 结合，提供了一种非常好的交互式数据绘图环境。

4. IPython

IPython 是 Python 科学计算标准工具的组成部分，它提供了一个强健而高效的环境。它是一个增强的 Python shell，可以提高编写、测试、调试 Python 代码的速度。它主要用于交互式数据处理和利用 Matplotlib 对数据进行可视化处理。

5. SciPy

SciPy 是一组专门解决科学计算中各种标准问题域的包的集合，主要包有如下几种。

- scipy. integrate：数值积分方程和微分方程求解器。
- scipy. linalg：线性代数方程和矩阵分解功能。
- scipy. optimize：函数优化器及等式根查找算法。
- scipy. signal：信号处理工具。
- scipy. sparse：稀疏矩阵和稀疏线性系统求解器。
- scipy. special：SPECFUN 的包装器。
- scipy. state：标准连续和离散概率分布、各种统计检验方法。

6. Scikit-learn

Scikit-learn 是一个强大的基于 Python 的机器学习数据挖掘算法库，封装了基本所有主流的机器学习算法。它使用方便，开源并且可供商业应用。它基于 BSD 开源许可证，依赖于 NumPy、SciPy、Matplotlib。Scikit-learn 在工业界已经被很多有数据挖掘需求的企业所采纳，它的基本功能主要被分为六个部分：分类(classification)、回归(regression)、聚类(clustering)、数据降维(dimensionality reduction)、模型选择(model selection)、数据预处理(preprocessing)。具体可以参考官方网站(http://scikit-learn. org/stable/)上的文档。

13.2.3 环境介绍

适合初学者的 Python 工具：Anaconda。Anaconda 里面集成了很多关于 Python 科学计算的第三方库，可以完全兼容 Python 2. 7 和 Python 3. 5，安装和使用都比较方便。不同的操作系统都可以直接在官方网站 https://www. continuum. io/downloads 选择对应的 Python 版本的安装包进行安装。下载完之后，可以按照 Anaconda 默认路径安装，也可以自己选定安装位置。Anaconda 安装时会自动把 bin 目录加入到环境变量 path 中。

13.3 数据预处理

现实世界中数据大多存在不完整、不一致、含噪声和维度高等问题，无法直接进行数据挖掘，或挖掘结果差强人意。为了提高数据挖掘的质量，产生了数据预处理技术。数据预处理主要包含数据清洗、数据集成、数据归约和数据变换等。这些数据处理技术在数据

挖掘之前使用，大大提高数据挖掘模式的质量，降低挖掘所需要的时间。

13.3.1　数据清洗

数据清洗是指发现并纠正数据文件中可识别的错误，包括检查数据一致性、处理无效值和缺失值等。数据缺失在大部分数据分析应用中都很常见，Pandas 使用浮点值 NaN 表示浮点和非浮点数组中的缺失数据，Python 内置的 None 值会被当作 NaN 处理。

【例 13.1】　判断是否存在缺失值。

程序如下：

```
from pandas import Series,DataFrame
string_data=Series(['abcd','efgh','ijkl','mnop'])
print(string_data)
print("..........\n")
string_data[0]=None
print(string_data.isnull())
```

程序运行结果如图 13.2 所示。

有些应用需要将缺失值设为常数，可以通过调用 fillna 将缺失值替换为常数值。

【例 13.2】　利用 fillna 参数实现替换缺失值。

程序如下：

```
from pandas import Series,DataFrame,np
from numpy import nan as NA
data=DataFrame(np.random.randn(7,3))
data.ix[:4,1]=NA
data.ix[:2,2]=NA
print(data)
print("..........")
print(data.fillna(0))
```

程序运行结果如图 13.3 所示。

```
0     abcd
1     efgh
2     ijkl
3     mnop
dtype: object
...........

0      True
1     False
2     False
3     False
dtype: bool
```

图 13.2　判断是否存在缺失值

```
          0         1         2
0 -0.800889       NaN       NaN
1  1.089445       NaN       NaN
2 -0.618349       NaN       NaN
3 -0.402036       NaN  0.856284
4  0.102542       NaN -0.885808
5 -0.735795  1.853092 -0.640952
6  1.304427  0.092933  0.193558
...........
          0         1         2
0 -0.800889  0.000000  0.000000
1  1.089445  0.000000  0.000000
2 -0.618349  0.000000  0.000000
3 -0.402036  0.000000  0.856284
4  0.102542  0.000000 -0.885808
5 -0.735795  1.853092 -0.640952
6  1.304427  0.092933  0.193558
```

图 13.3　利用 fillna 参数实现替换缺失值

【例 13.3】 检测和过滤异常值。

程序如下：

```
from numpy import nan as NA
data= DataFrame(np.random.randn(1000,4))
print(data.describe())
print("\n...找出某一列中绝对值大小超过 3 的项...\n")
col= data[3]
print(col[np.abs(col) > 3])
print("\n...找出全部绝对值超过 3 的值的行...\n")
print(col[(np.abs(data) > 3).any(1)])
```

程序运行结果如图 13.4 所示。

```
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.021607     0.010721     0.036979    -0.010430
std       1.010394     1.031516     0.984209     1.039357
min      -3.129855    -2.730729    -3.697866    -3.187682
25%      -0.739682    -0.678421    -0.630820    -0.710234
50%      -0.070473     0.032551     0.112817    -0.003704
75%       0.712484     0.723364     0.734883     0.679854
max       3.171343     3.788392     2.830543     3.494408

....找出某一列中绝对值大小超过3的项...

17     3.494408
284   -3.024707
492   -3.187682
Name: 3, dtype: float64

....找出全部绝对值超过3的值的行...

17     3.494408
97     1.386348
195   -0.140969
204    0.415686
284   -3.024707
354    1.086080
492   -3.187682
647    0.395741
Name: 3, dtype: float64
```

图 13.4 检测和过滤异常值

13.3.2 数据变换

数据变换将数据转换或者统一为适合进行数据挖掘的形式。例如，有的数据挖掘算法只能处理离散型数据，这时需要将连续数据离散化。假设有一组人员年龄数据，需要将它们划分为不同的年龄组，将这些数据划分为 18～25(Youth)、26～35(YoungAdult)、35～60(MiddleAged)以及 60 以上(Senior)四个面元。可以使用 Pandas 的 cut()函数实现面元划分。

【例 13.4】 面元划分。

程序如下：

```
import pandas as pd
import numpy as np
ages = [20,22,25,27,21,23,37,31,61,45,41,32]
```

```
bins = [18,25,35,60,100]
group_names = ['Youth','YoungAdult','MiddleAged','Senior']                    #设置面元名称
print(pd.cut(ages,bins,labels=group_names))
print(cats)
```

程序运行结果如图 13.5 所示。

```
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

图 13.5　面元划分

【例 13.5】 将均匀分布的数据分成四组。

程序如下：

```
import pandas as pd
import numpy as np
data = np.random.rand(20)
print(pd.cut(data,4,precision=2) (0.037,0.26])
```

程序运行结果如图 13.6 所示。

```
[(0.23, 0.44], (0.021, 0.23], (0.44, 0.65], (0.65, 0.86], (0.65, 0.86], ..., (0.65, 0.86], (0.021, 0.23], (0.44, 0.65], (0.021, 0.23], (0.02
1, 0.23]]
Length: 20
Categories (4, object): [(0.021, 0.23] < (0.23, 0.44] < (0.44, 0.65] < (0.65, 0.86]]
```

图 13.6　将均匀分布的数据分成四组

13.3.3　数据集成

数据集成是把不同来源、格式、特点、性质的数据在逻辑上或物理上有机地集中，为提供全面的数据共享做准备。数据集的合并(merge)或连接(join)运算是关系型数据库的核心运算。Pandas 的 merge()函数提供了数据合并功能。

【例 13.6】 调用 merge()进行数据合并。

程序如下：

```
import pandas as pd
import numpy as np
df1 = pd.DataFrame({'key': ['b','b','a','c','a','a','b'],'data1': range(7)})
df2 = pd.DataFrame({'key': ['a','b','d'],'data2': range(3)})
print(pd.merge(df1,df2,on='key'))            # on 指定合并列,若无指定,将重叠列的列当作合并列
```

程序运行结果如图 13.7 所示。

```
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

图 13.7　数据合并

默认情况下，merge()做的是内合并；关系数据库中除了内合并之外还有左外合并、右外合并和全外合并。

【例 13.7】 调用 merge()进行数据左外合并。

程序如下：

```
import pandas as pd
```

```
import numpy as np
pd.merge(df1,df2,how='outer'):
df1 =pd.DataFrame({'key': ['b','b','a','c','a','b'],'data1': range(6)})
df2 =pd.DataFrame({'key': ['a','b','a','b','d'],'data2': range(5)})
print(pd.merge(df1,df2,on='key',how='left'))
```

程序运行结果如图 13.8 所示。

	data1	key	data2
0	0	b	1.0
1	0	b	3.0
2	1	b	1.0
3	1	b	3.0
4	2	a	0.0
5	2	a	2.0
6	3	c	NaN
7	4	a	0.0
8	4	a	2.0
9	5	b	1.0
10	5	b	3.0

图 13.8 df1 和 df2 左外合并

13.3.4 数据归约

数据归约技术可以用来得到数据集的归约表示，产生更小且保持数据完整性的新数据集。其意义在于降低无效、错误数据，降低存储成本。少量且具有代表性的数据可以加快数据挖掘速度。下面以数据归约中的直方图为例说明数据归约。

【例 13.8】 数据直方图。

程序如下：

```
import pandas as pd
import numpy as np
x=[1,1,5,5,5,5,8,8,10,10,10,10,14,14,14,14,15,15,15,15,15,15,18,18,18,18,18,18,18,18,18,20,2,20,20,20,
20,20,20,21,21,21,25,25,25,25,25,28,28,30,30,30]
x=pd.Series(x)
count=x.value_counts()
count.plot(kind='bar')
plt.show()
```

程序运行结果如图 13.9 所示。

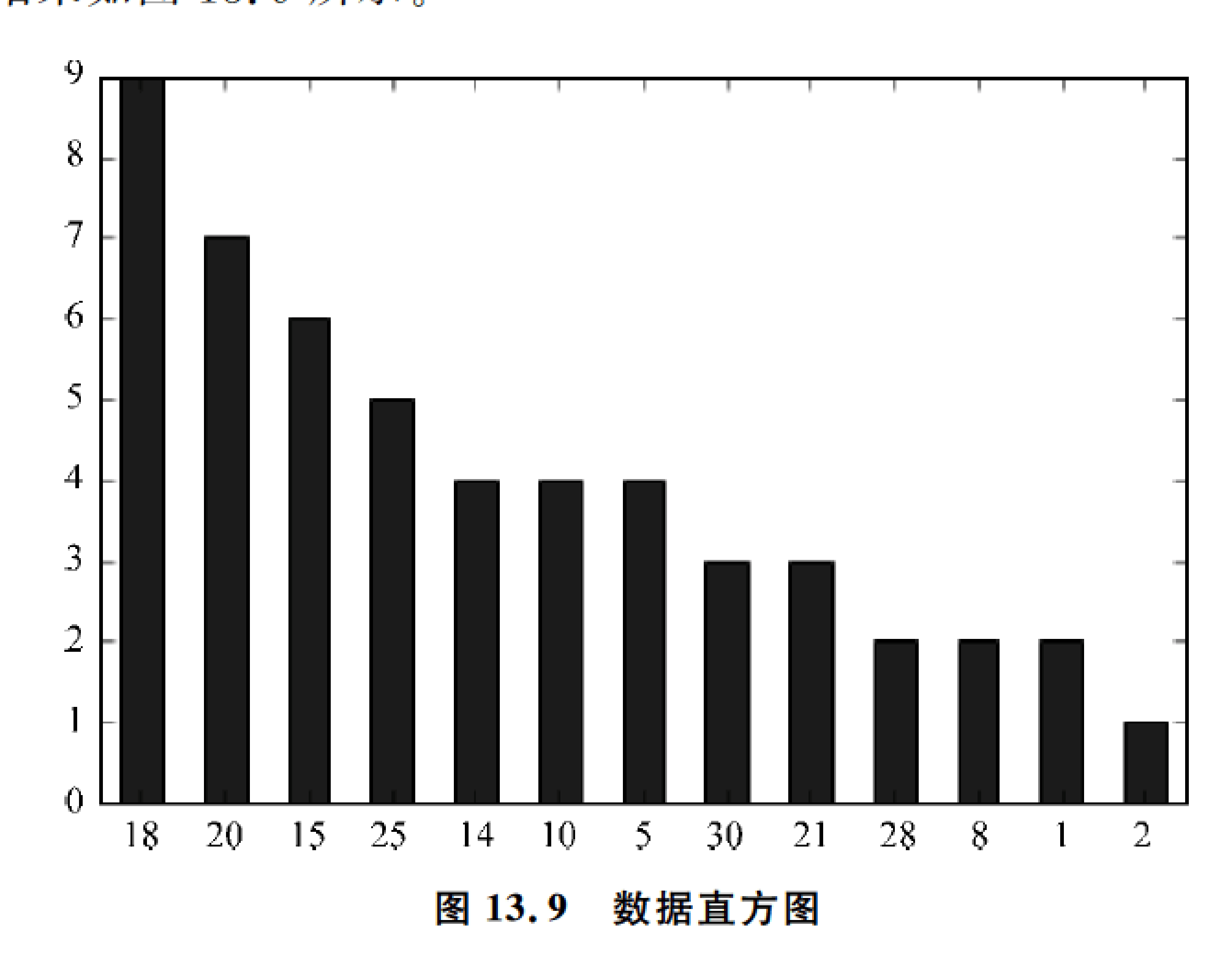


图 13.9 数据直方图

13.4 聚类分析

13.4.1 关于聚类分析

聚类分析是指根据“物以类聚”原理，将本身没有类别的样本聚集成不同的类别的分析过程。聚类分析的准则是使属于同一个类中的对象尽可能相似，而不同类间的对象尽可能不相似。聚类分析没有预定义的类，属于无监督学习。典型的聚类算法有 K-means 算法、DBSCAN 算法、CLARANS 算法和 BIRCH 算法等。本节以 K-means 算法为例说明聚类过程。

13.4.2 K-means 算法

K-means 算法是一种经典的聚类算法，在模式识别中得到了广泛的应用。算法中有两个关键问题需要考虑：一个是如何评价对象的相似性，通常用距离来度量，距离越近越相似；另外一个是如何评价聚类的效果，通常采用误差平方和函数来作为评价准则。

1. 欧氏距离

K-means 算法对初始聚类中心较敏感，相似度计算方式会影响聚类的划分。常见的相似度计算方法是欧式距离。假设给定的数据集 $X=\{x_m \mid m=1,2,\cdots,\text{total}\}$，$X$ 中的样本用 d 个描述属性 $A_1,A_2\cdots A_d$（维度）来表示。

数据样本 $x_i=(x_{i_1},x_{i_2},\cdots,x_{i_d})$，$x_j=(x_{j_1},x_{j_2},\cdots,x_{j_d})$其中，$x_{i_1},x_{i_2},\cdots,x_{i_d}$ 和 x_{j_1}，$x_{j_2},\cdots,x_{j_d}$ 分别是样本 x_i 和 x_j 对应的 d 个描述属性 $A_1,A_2,\cdots,A_d$ 的具体取值。

样本 x_i 和 x_j 之间的相似度通常用它们之间的距离 $d(x_i,x_j)$ 来表示。距离越小，样本 x_i 和 x_j 越相似，差异度越小；距离越大，样本 x_i 和 x_j 越不相似，差异度越大。

欧式距离公式如下：

$$d(x_i,x_j)=\sqrt{\sum_{k=1}^{d}(x_{ik}-x_{jk})^2}$$

2. 评价准则

K-means 聚类算法使用误差平方和函数来评价聚类性能。给定数据集 X，其中只包含描述属性，不包含类别属性。假设 X 包含 K 个聚类子集 $X_1,X_2,\cdots,X_K$；各个聚类子集中的样本数量分别为 $n_1,n_2,\cdots,n_K$；各个聚类子集的均值代表点（也称聚类中心）分别为 $m_1,m_2,\cdots,m_K$。

误差平方和准则函数公式为：

$$E=\sum_{i=1}^{K}\sum_{p\in X_i}\|p-m_i\|^2$$

误差平方和越小，聚类效果越好。

3. 算法过程

K-means 算法的输入量是聚类个数 K，将 n 个数据对象划分为 K 个聚类，所获得的聚

类满足：同一聚类中的对象相似度较高，而不同聚类中的对象相似度较小。聚类相似度是利用各聚类中对象的均值所获得的一个聚类中心来进行计算的。算法过程如图 13.10 所示。

算法：K-means算法。
输入：簇的数目K和包含n个对象的数据库。
输出：K个簇，使平方误差和最小。
算法步骤：
(1) 为每个聚类确定一个初始聚类中心，这样就有K个初始聚类中心。
(2) 将样本集中的样本按照最小距离原则分配到最邻近聚类。
(3) 使用每个聚类中的样本均值作为新的聚类中心。
(4) 重复步骤(2)和步骤(3)，直到聚类中心不再变化。
(5) 结束，得到K个聚类。

图 13.10 K-means 算法过程

算法流程图如图 13.11 所示。

【例 13.9】 利用 Python 将数据集 testSet.txt 中的数据(二维坐标点)聚为四类。testSet.txt 如图 13.12 所示，共有 20 组数据，为 float 类型。

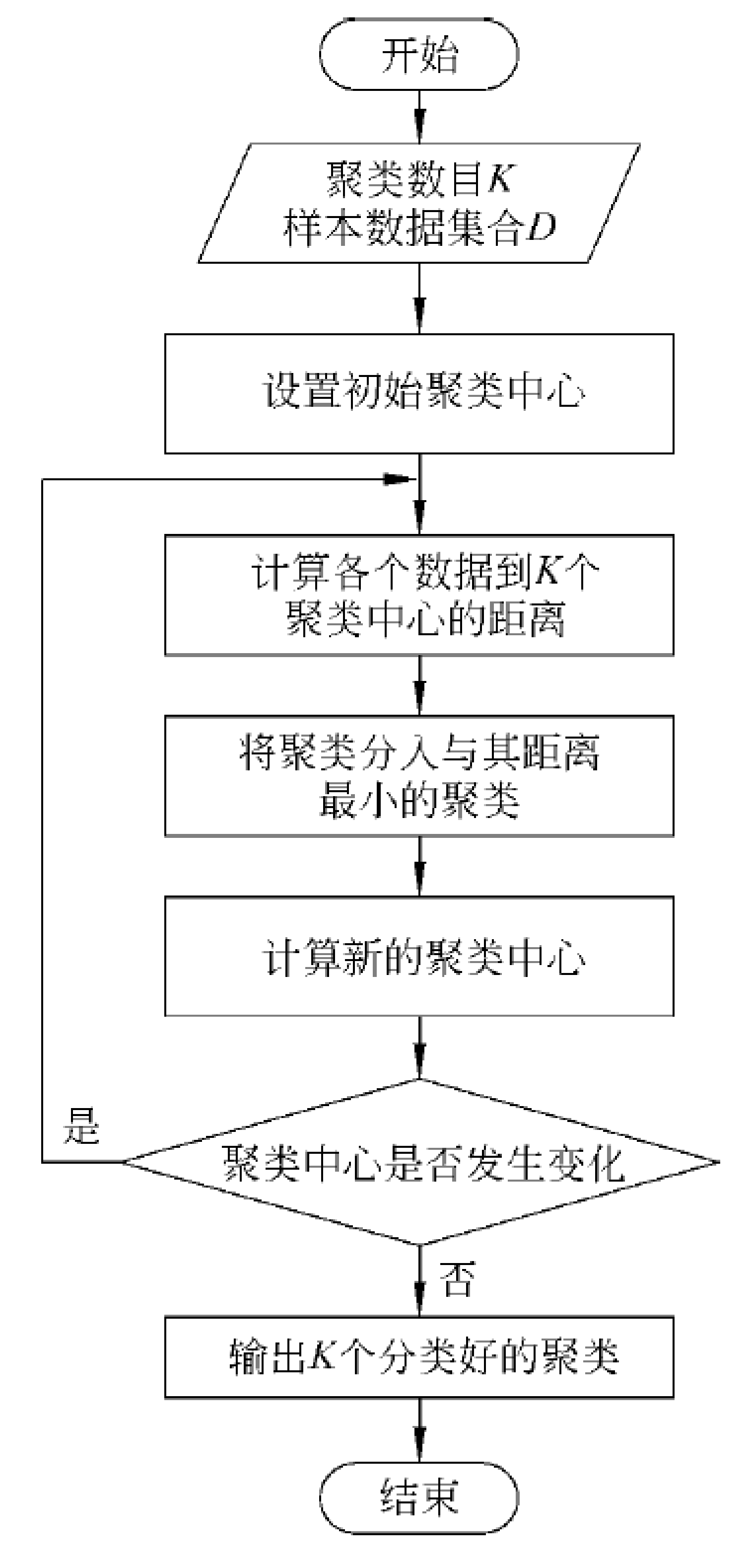


图 13.11 K-means 算法流程图

1.658985	4.285136
-3.453687	3.424321
4.838138	-1.151539
-5.379713	-3.362104
0.972564	2.924086
-3.567919	1.531611
0.450614	-3.302219
-3.487105	-1.724432
2.668759	1.594842
-3.156485	3.191137
3.165506	-3.999838
-2.786837	-3.099354
4.208187	2.984927
-2.123337	2.943366
0.704199	-0.479481
-0.392370	-3.963704
2.831667	1.574018
-0.790153	3.343144
2.943496	-3.357075
-3.195883	-2.283926

图 13.12 testSet.txt 数据

程序如下：

```
#首先导入所需要的库
```

```
import numpy
import random
import codecs
import copy
import re
import matplotlib.pyplot as plt
#其次计算向量 vec1 和向量 vec2 之间的欧氏距离
def calcuDistance(vec1,vec2):
    return numpy.sqrt(numpy.sum(numpy.square(vec1-vec2)))
#载入数据测试数据集,数据由文本保存,为二维坐标
def loadDataSet(inFile):
    inDate = codecs.open(inFile,'r','utf-8').readlines()
    dataSet = list()
    for line in inDate:
        line = line.strip()
        strList = re.split('[ ]+',line)     #去除多余的空格
                                            #print strList[0],strList[1]
        numList = list()
        for item in strList:
            num = float(item)
            numList.append(num)
                                            #print numList
        dataSet.append(numList)

    return dataSet
                                            #初始化 k 个聚类中心,随机获取
def initCentroids(dataSet,k):
    return random.sample(dataSet,k)         #从 dataSet 中随机获取 k 个数据项返回
#对每个属于 dataSet 的 item,计算 item 与 centroidList 中 k 个聚类中心的欧式距离,找出
#距离最小的,并将 item 加入相应的簇中
def minDistance(dataSet,centroidList):
    clusterDict = dict()                    #用 dict 来保存聚类结果
    for item in dataSet:
        vec1 = numpy.array(item)            #转换成 array 形式
        flag = 0                            #簇分类标记,记录与相应簇距离最近的那个簇
        minDis = float("inf")               #初始化为最大值
        for i in range(len(centroidList)):
            vec2 = numpy.array(centroidList[i])
            distance = calcuDistance(vec1,vec2)  #计算相应的欧式距离
            if distance < minDis:
                minDis = distance
                flag = i        #循环结束时,flag 保存的是与当前 item 距离最近的那个簇标记
        if flag not in clusterDict.keys():  #簇标记不存在,进行初始化
            clusterDict[flag] = list()
```

```
                                        #print flag,item
            clusterDict[flag].append(item)  #加入相应的类别中
        return clusterDict                  #返回新的聚类结果
#计算每列的均值,即找到聚类中心
def getCentroids(clusterDict):
                                        #得到 k个质心
    centroidList = list()
    for key in clusterDict.keys():
        centroid = numpy.mean(numpy.array(clusterDict[key]),axis = 0)
        centroidList.append(centroid)
    return numpy.array(centroidList).tolist()
#计算簇集合间的均方误差,将簇类中各个向量与质心的距离进行累加求和
def getVar(clusterDict,centroidList):
    sum = 0.0
    for key in clusterDict.keys():
        vec1 = numpy.array(centroidList[key])
        distance = 0.0
        for item in clusterDict[key]:
            vec2 = numpy.array(item)
            distance+ = calcuDistance(vec1,vec2)
        sum+ = distance
    return sum

#展示聚类结果
def showCluster(centroidList,clusterDict):
    colorMark = ['or','ob','og','ok','oy','ow']
    #不同簇类的标记,'or'-->'o'代表圆形,'r'代表 red,'b':blue
    centroidMark = ['dr','db','dg','dk','dy','dw']  #聚类中心标记 同上'd'代表菱形
    for key in clusterDict.keys():
        plt.plot(centroidList[key][0],centroidList[key][1],centroidMark[key],markersize = 12) #画聚类中心点
        for item in clusterDict[key]:
            plt.plot(item[0],item[1],colorMark[key]) #画簇类下的点
    plt.show()

if __name__ == '__main__':
    inFile = "F:/python/testSet.txt"              #数据集文件
    dataSet = loadDataSet(inFile)                 #载入数据集
    centroidList = initCentroids(dataSet,4)       #初始化质心,设置 k=4
    clusterDict = minDistance(dataSet,centroidList) #第一次聚类迭代
    newVar = getVar(clusterDict,centroidList)
                                    #获得均方误差值,通过新旧均方误差来获得迭代终止条件
    oldVar = -0.0001                              #旧均方误差值初始化为-1
    print ('***** 第一次迭代 ***** ')
    print ( )
```

```
print ('簇类')
for key in clusterDict.keys():
    print (key,'-->',clusterDict[key])
print ('k个均值向量:',centroidList)
print ('平均均方误差:',newVar)
print ( )
showCluster(centroidList,clusterDict)                 #展示聚类结果
k=2
while abs(newVar-oldVar) >=0.0001:             #当连续两次聚类结果小于 0.0001 时,迭代结束
    centroidList =getCentroids(clusterDict)            #获得新的质心
    clusterDict =minDistance(dataSet,centroidList)     #新的聚类结果
    oldVar =newVar
    newVar =getVar(clusterDict,centroidList)

    print ('*****第%d次迭代*****' %k)
    print( )
    print ('簇类')
    for key in clusterDict.keys():
        print (key,'-->',clusterDict[key])
    print('k个均值向量:',centroidList)
    print('平均均方误差:',newVar)
    print( )
    showCluster(centroidList,clusterDict)             #展示聚类结果
    k+=1
```

【结果分析】

进行了四次迭代,第一次迭代结果如图 13.13 所示。

簇 1:[[1.658985,4.285136],[0.972564,2.924086],[2.668759,1.594842],[4.208187,2.984927],[-2.123337,2.943366],[0.704199,-0.479481],[2.831667,1.574018],[-0.790153,3.343144]]

簇 2:[[-3.453687,3.424321],[-5.379713,-3.362104],[-3.567919,1.531611],[-3.487105,-1.724432],[-3.156485,3.191137],[-2.786837,-3.099354],[-0.39237,-3.963704],[-3.195883,-2.283926]]

簇 3:[[4.838138,-1.151539]]

簇 4:[[0.450614,-3.302219],[3.165506,-3.999838],[2.943496,-3.357075]]

均值向量:[[2.668759,1.594842],[-3.195883,-2.283926],[3.165506,-3.999838],[4.838138,-1.151539]]

均方误差:44.7305422925

第二次迭代结果如图 13.14 所示。

簇 1:[[1.658985,4.285136],[0.972564,2.924086],[2.668759,1.594842],[4.208187,2.984927],[-2.123337,2.943366],[0.704199,-0.479481],[2.831667,1.574018],[-0.790153,3.343144]]

簇 2:[[-3.453687,3.424321],[-5.379713,-3.362104],[-3.567919,1.531611],[-3.487105,-1.724432],[-3.156485,3.191137],[-2.786837,-3.099354],[-3.195883,-2.283926]]

簇 3:[4.838138,-1.151539]]

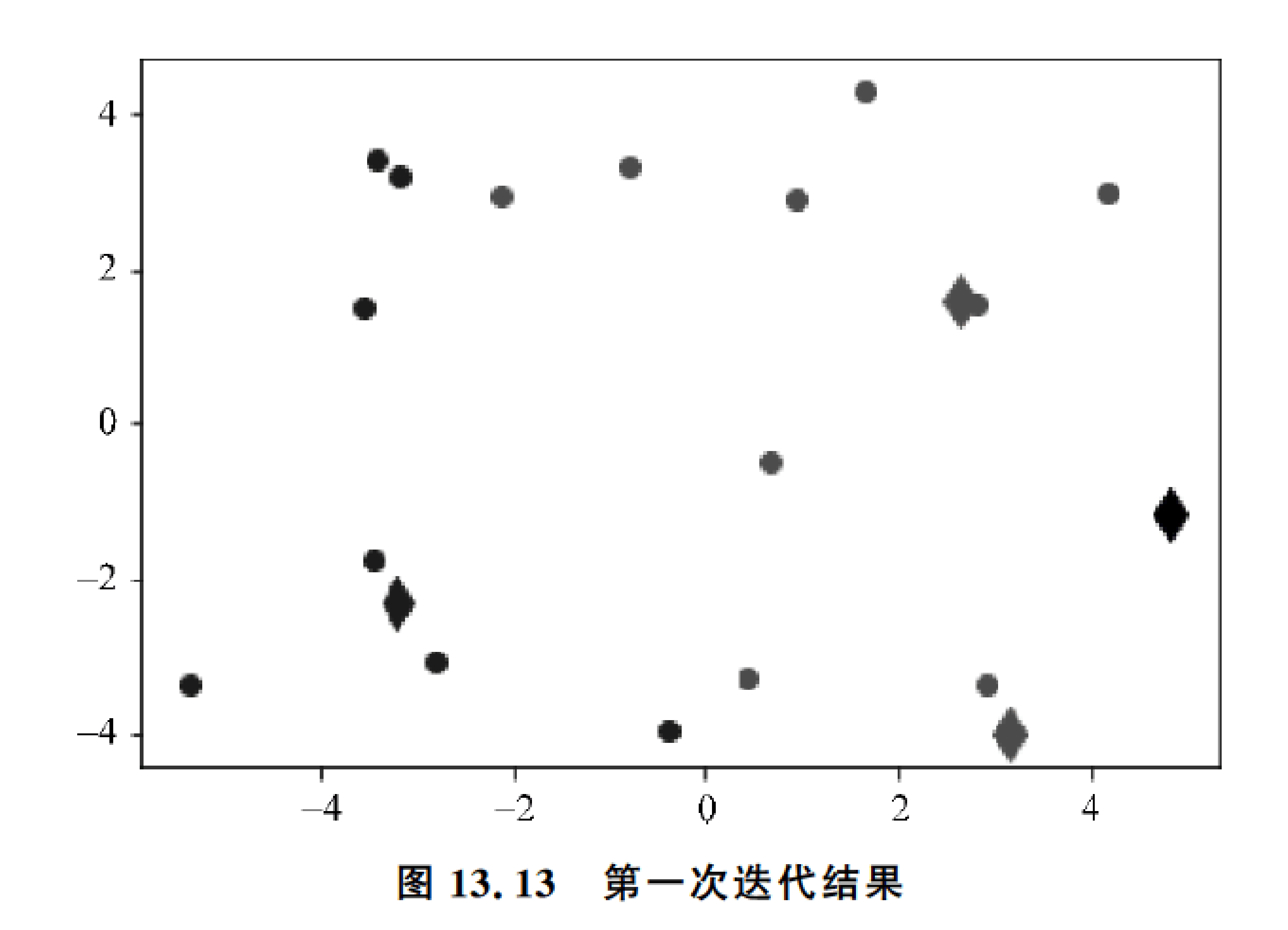


图 13.13　第一次迭代结果

簇 4：[[0.450614,− 3.302219],[3.165506,− 3.999838],[− 0.39237,− 3.963704],[2.943496,− 3.357075]]

均值向量：[[1.2663588749999999,2.39625475],[− 3.1774998749999999, − 0.7858063750000001],[4.838138,− 1.151539],[2.1865386666666667,− 3.553044]]

均方误差：42.5363920697

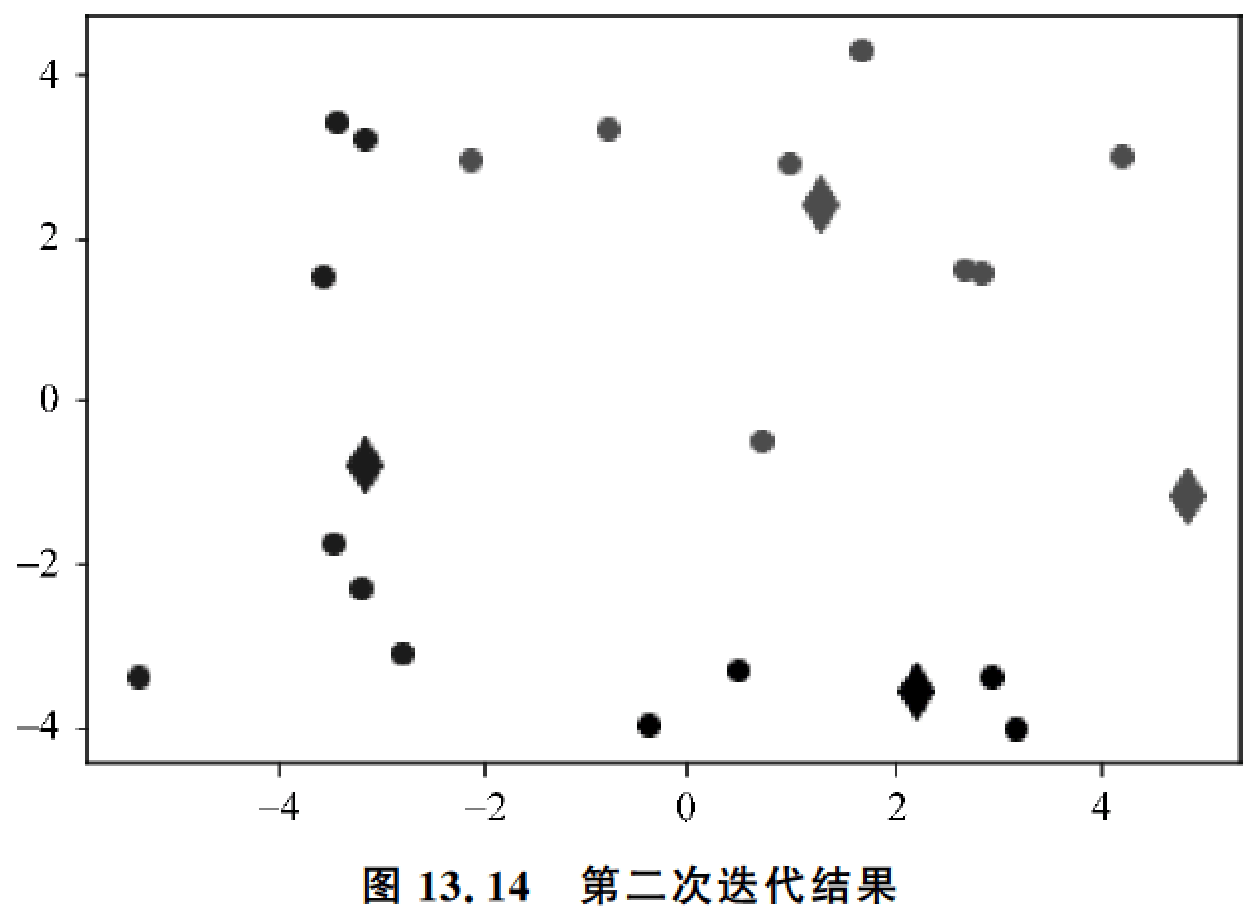


图 13.14　第二次迭代结果

第三次迭代结果如图 13.15 所示。

簇 1：[[1.658985,4.285136],[0.972564,2.924086],[2.668759,1.594842],[4.208187,2.984927],[− 2.123337,2.943366],[0.704199,− 0.479481],[2.831667,1.574018],[− 0.790153,3.343144]]

簇 2：[[− 3.453687,3.424321],[− 5.379713,− 3.362104],[− 3.567919,1.531611],[− 3.487105,− 1.724432],[− 3.156485,3.191137],[− 2.786837,− 3.099354],[− 3.195883,− 2.283926]]

簇 3：[[4.838138,− 1.151539]]

簇 4：[[0.450614,− 3.302219],[3.165506,− 3.999838],[− 0.39237, − 3.963704],[2.943496,− 3.357075]]

均值向量：[[1.2663588749999999,2.39625475],[− 3.5753755714285704, − 0.3318210000000003],[4.

838138,-1.151539],[1.5418115000000001,-3.655709]]

均方误差：42.7009544228

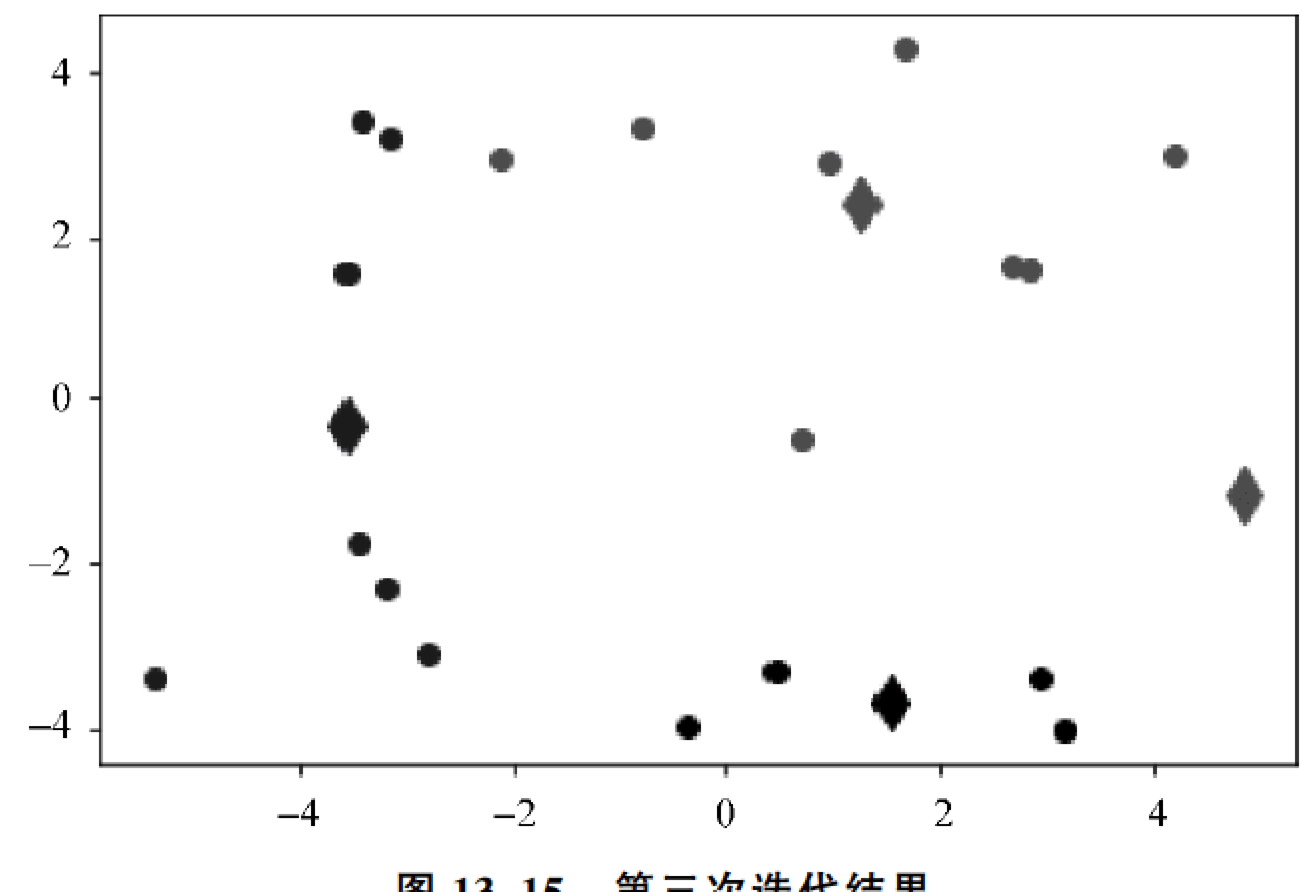


图 13.15　第三次迭代结果

第四次迭代结果如图 13.16 所示。

簇 1：[[1.658985,4.285136],[0.972564,2.924086],[2.668759,1.594842],[4.208187,2.984927],[-2.123337,2.943366],[0.704199,-0.479481],[2.831667,1.574018],[-0.790153,3.343144]]

簇 2：[[-3.453687,3.424321],[-5.379713,-3.362104],[-3.567919,1.531611],[-3.487105,-1.724432],[-3.156485,3.191137],[-2.786837,-3.099354],[-3.195883,-2.283926]]

簇 3：[[4.838138,-1.151539]]

簇 4：[[0.450614,-3.302219],[3.165506,-3.999838],[-0.39237,-3.963704],[2.943496,-3.357075]]

均值向量：[[1.2663588749999999,2.39625475],[-3.5753755714285704,-0.33182100000000003],[4.838138,-1.151539],[1.5418115000000001,-3.655709]]

均方误差：42.7009544228

由上可见，当聚类中心和误差不再变化时，聚类结束。

4. K-means 算法优缺点

K-means 算法是解决聚类问题的一种经典算法，该算法简单、快速。对处理大数据集，该算法具有可伸缩和高效率的优点。其时间复杂度是 $O(n*K*t)$，其中，n 是所有对象的数目，K 是簇的数目，t 是迭代的次数。通常 $K<<n$ 且 $t<<n$。

但是 K-means 算法在簇的平均值被定义的情况下才能使用，这对于处理符号属性的数据不适用；同时必须事先给出簇的数目 K，而且对初值敏感。

13.5　分　　类

13.5.1　关于分类

分类是根据已知的样本数据得出分类函数或模型，来判断其他未知数据的类别。决

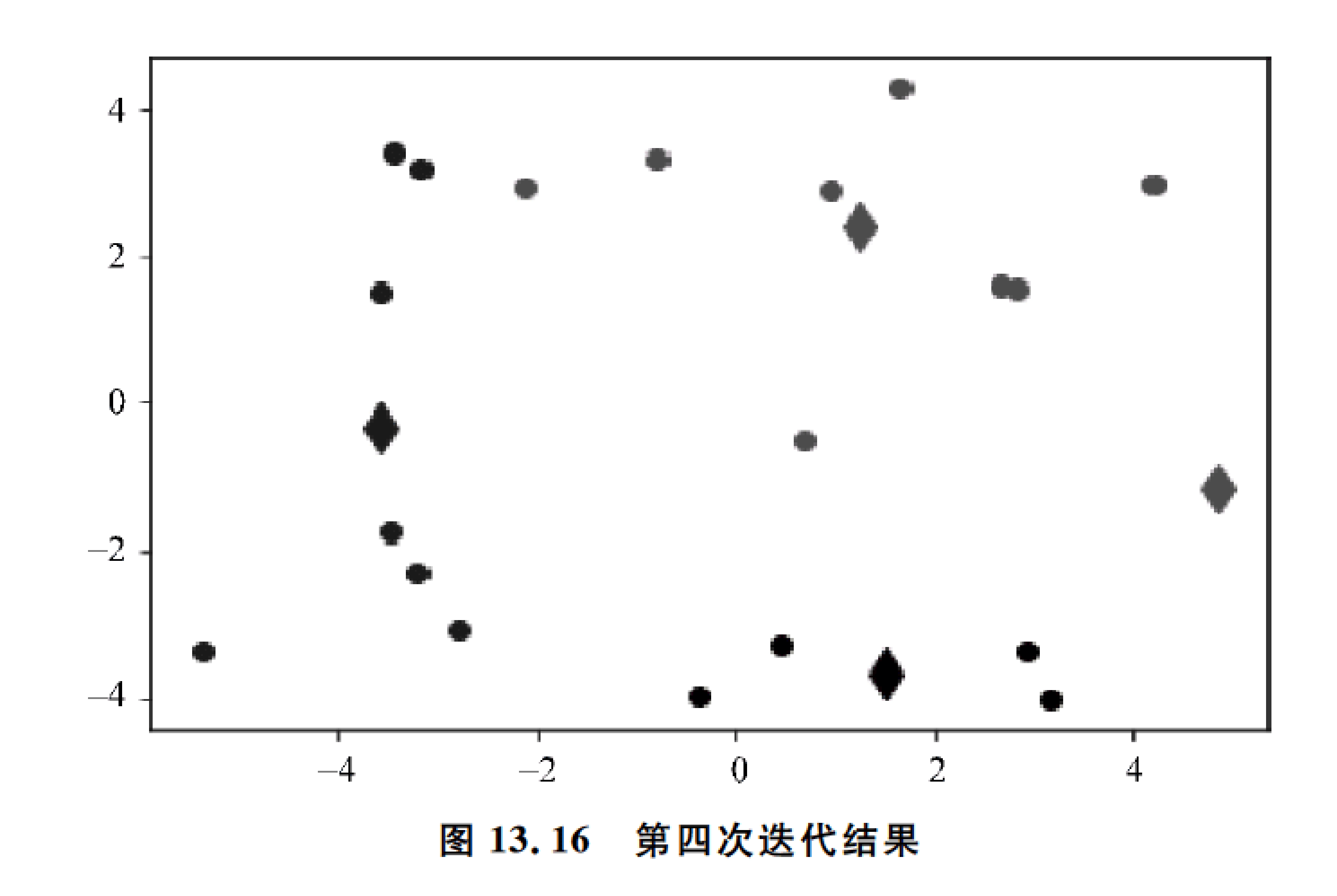


图 13.16　第四次迭代结果

策树分类算法是常用的分类算法之一。决策树分类算法自 20 世纪 60 年代以来，在分类、预测、规则提取等领域有着广泛的应用。而自从决策树经典分类算法——ID3 算法提出以后，决策树算法在机器学习、知识发现等领域得到了进一步的应用和巨大的发展。

13.5.2　分类相关概念

1．数据对象、属性

数据集由数据对象组成。数据集中的数据对象是实体。

属性是一个数据字段，表示数据对象的一个特征。每个数据对象都有若干个属性组成。

2．分类器

分类的关键是找出一个合适的分类函数或分类模型。分类的过程是依据已知的数据样本建立一个分类函数或者构造出一个分类模型，即分类器。该函数或模型能够把数据库中的数据对象映射到某一个给定的类别中，从而判断数据对象的类别。

3．训练集

分类的训练集也称样本数据，是构造分类器的基础。训练集是由特殊要求的数据对象组成，它的每个对象的所属类别已知。在构造分类器时，需要输入一定量的训练集。选取的训练集是否合适，直接影响到构造的分类器性能的好坏。

4．测试集

与训练集一样，测试集也是由类别属性已知的数据对象组成。测试集是用来测试基于训练集构成的分类器性能。在决策树生成后，测试集依次输入决策树。用决策树判定的测试集对象的所属类别与测试集已知所属类别进行比较，从而得出决策树的正确率。

5．信息论相关定义

ID3 算法是以信息论为基础，以信息熵和信息增益度为衡量标准，从而实现对数据的归纳分类。以下是一些信息论的基本概念。



定义 13.1　熵：系统存在的一个状态函数，泛指某些物质系统状态的一种度量。信息熵在信息论中称为平均信息量，是对被传送信息进行度量所采用的一种平均值。

定义 13.2　若存在 n 个相同概率的消息，则每个消息的概率 p 是 $1/n$，一个消息传递的信息量为 $-\log_2(p)$。

定义 13.3　若有 n 个消息，其给定概率分布为 $P=(p_1,p_2,\cdots,p_n)$，则由该分布传递的信息量称为 P 的熵，记为

$$I=-\sum_{i=1}^{n}P_i\log_2 P_i$$

定义 13.4　若一个集合 D 先根据非类别属性 X 的值将其分成集合 $d_1,d_2,\cdots,d_n$，再根据类别属性的值被分成互相独立的类 $c_{1j},c_{2j},\cdots,c_{ij}$，则识别 D 的一个元素所属的类所需要的信息量为 $\text{info}(D)=I(P)$，其中 $P=(|c_{1j}|/|d_j|,\cdots,|c_{ij}|/|d_j|)$。

定义 13.5　若先根据非类别属性 X 的值将 D 分成集合 $d_1,d_2,\cdots,d_n$，则确定 D 中一个元素类的信息量可通过确定 d_i 的加权平均值来得到，即 $\text{info}(d_i)$ 的加权平均值为

$$\text{info}(X,D)=\sum_{j=1}^{n}(d_j/D)\text{info}(d_j)$$

定义 13.6　信息增益度是两个信息量之间的差值，其中一个信息量是需确定 D 的一个元素的信息量，另一个信息量是在已得到的属性 X 的值后需确定的 D 的一个元素的信息量。信息增益度公式为

$$\text{Gain}(X,D)=\text{info}(D)-\text{info}(X,D)$$

13.5.3　ID3 算法

ID3 算法的核心思想是通过计算属性的信息增益来选择决策树各级结点上的分类属性，使得在每一个非叶子结点都可获得被测样本的最大类别信息。其基本方法是：计算所有的属性的信息增益，选择其中最大的属性作为分裂属性产生决策树结点，然后根据该属性的不同取值建立相同数量的分支，再对各分支递归调用该方法来建立分支，直到子集中仅包括同一类别或没有可分裂的属性为止。由此可以得到一棵决策树，来对样本进行分析预测。

ID3 算法流程图如图 13.17 所示。

【例 13.10】　使用 ID3 算法生成决策树。数据集如表 13.1 所示。

表 13.1　数据集

RID	age	income	student	Credit_rating	Class：buy_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes

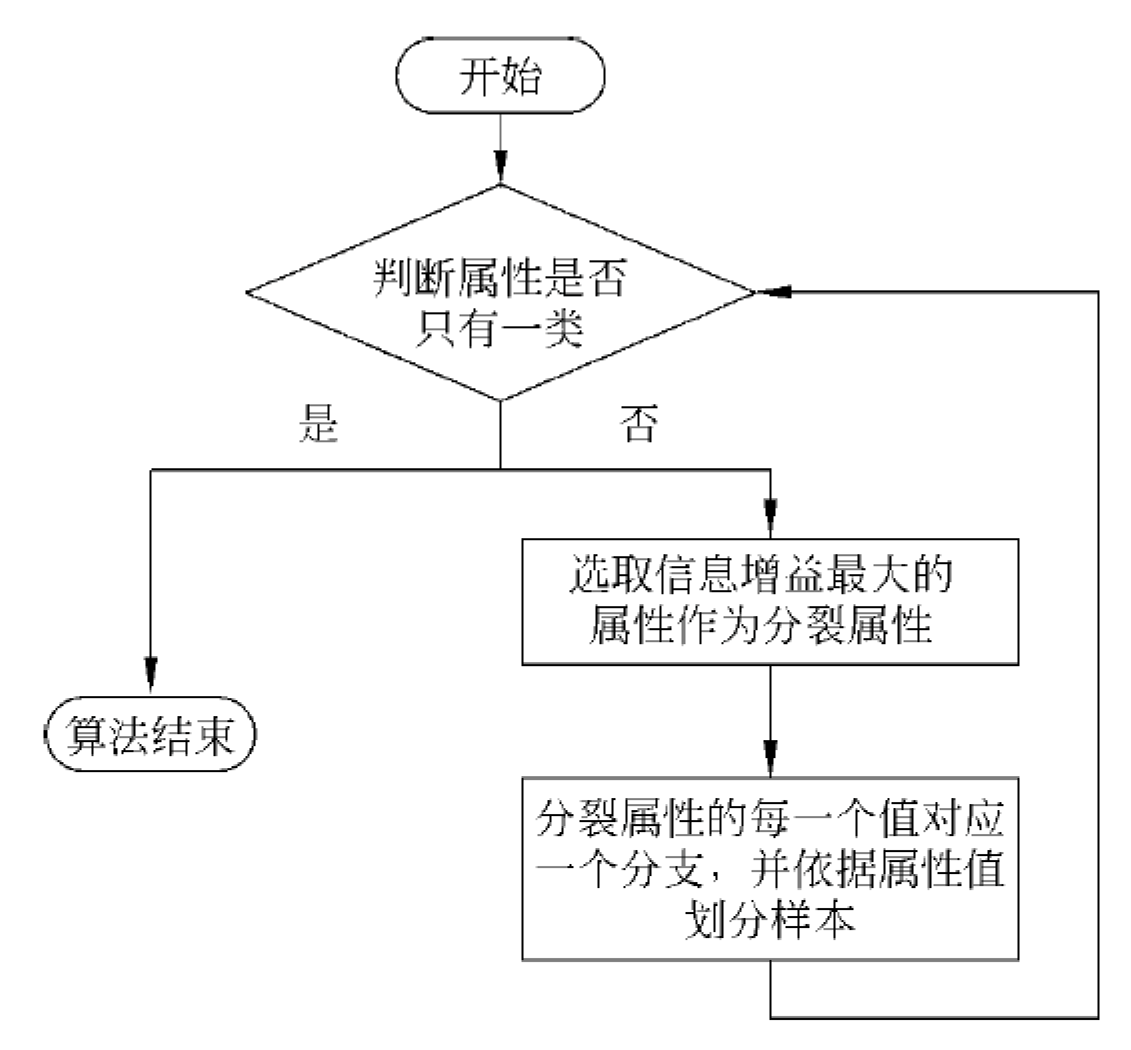


图 13.17　ID3 算法流程图

6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

程序如下：

```
#首先创建数据集
def createDataSet():
    dataSet=[['youth','high','no','fair','no'],
             ['youth','high','no','excellent','no'],
             ['middle_aged','high','no','fair','yes'],
             ['senior','medium','no','fair','yes'],
             ['senior','low','yes','fair','yes'],
             ['senior','low','yes','excellent','no'],
             ['middle_aged','low','yes','excellent','yes'],
             ['youth','medium','no','fair','no'],
             ['youth','low','yes','fair','yes'],
             ['senior','medium','yes','fair','yes'],
```

```
                ['youth','medium','yes','excellent','yes'],
                ['middle_aged','medium','no','excellent','yes'],
                ['middle_aged','high','yes','fair','yes'],
                ['senior','medium','no','excellent','no']]
    labels = ['age','income','student','credit_rating']
    return dataSet,labels
#计算数据集的信息熵
def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for feaVec in dataSet:
        currentLabel = feaVec[-1]
        if currentLabel not in labelCounts:
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel]+=1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt-=prob * log(prob,2)
return shannonEnt
#分割数据集
def splitDataSet(dataSet,axis,value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] ==value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet
#计算条件熵
def calcConditionalEntropy(dataSet,i,featList,uniqueVals):
    ce= 0.0
    for value in uniqueVals:
        subDataSet= splitDataSet(dataSet,i,value)
        prob= len(subDataSet) / float(len(dataSet))  #极大似然估计概率
        ce+=prob * calcShannonEnt(subDataSet)        #∑pH(Y|X=xi) 条件熵的计算
    return ce
#计算信息增益
def calcInformationGain(dataSet,baseEntropy,i):

    featList = [example[i] for example in dataSet]   #第 i维特征列表
    uniqueVals = set(featList)                       #转换成集合
    newEntropy = calcConditionalEntropy(dataSet,i,featList,uniqueVals)
    infoGain = baseEntropy- newEntropy          #信息增益,就是熵的减少,也就是不确定性的减少
```

```
        return infoGain
#使用 ID3 算法
def chooseBestFeatureToSplitByID3(dataSet):
    numFeatures = len(dataSet[0]) - 1                #最后一列是分类
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0
    bestFeature = -1
    for i in range(numFeatures):                     #遍历所有维度特征
        infoGain = calcInformationGain(dataSet,baseEntropy,i)
        if (infoGain > bestInfoGain):                #选择最大的信息增益
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature                               #返回最佳特征对应的维度
#因为递归构建决策树是根据属性的消耗进行计算的,所以可能会存在最后属性用完了,但是分类
#还是没有算完,这时候就会采用多数表决的方式计算结点分类
def majorityCnt(classList):
    classCount= {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote]= 0
        classCount[vote]+ = 1
    return max(classCount)
def createTree(dataSet,labels):
    classList= [example[- 1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):          #类别相同则停止划分
        return classList[0]
    if len(dataSet[0])==1:                                       #所有特征已经用完
        return majorityCnt(classList)
    bestFeat= chooseBestFeatureToSplitByID3(dataSet)
    bestFeatLabel= labels[bestFeat]
    myTree= {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues= [example[bestFeat] for example in dataSet]
    uniqueVals= set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]                          #为了不改变原始列表的内容进行了复制
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet,
                                        bestFeat,value),subLabels)
    return myTree
#可视化
def main():
    data,label = createDataSet()
    t1 = time.clock()
    myTree = createTree(data,label)
```

```
    t2 = time.clock()
    treePlotter.createPlot(myTree)
    print (myTree)
    print ('execute for ',t2- t1)
if __name__=='__main__':
main()
```

最后输出的决策树效果如图 13.18 所示。

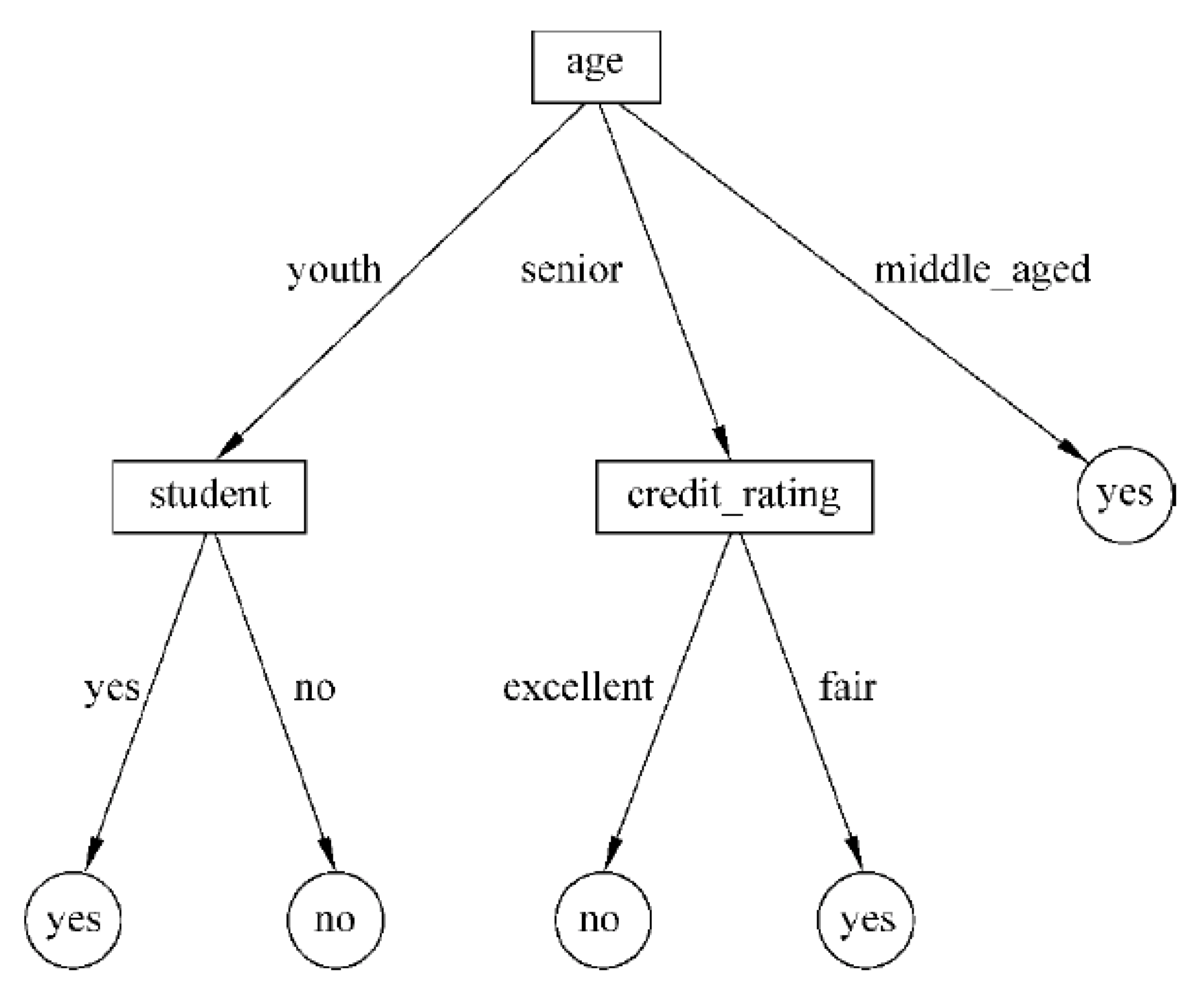


图 13.18 决策树效果

程序运行结果：

```
{'age': {'youth': {'student': {'yes': 'yes','no': 'no'}},'senior': {'credit_ rating': {'excellent': 'no
','fair': 'yes'}},'middle_aged': 'yes'}}
execute for   0.0004744615730487567
```

ID3 算法理论清晰，方法简单，生成的规则易被人理解。同时 ID3 算法不存在无解的危险，并且全盘使用训练数据，便于产生较为优化的决策树。

虽然 ID3 算法在选择属性时倾向于选择那些拥有多个属性值的属性作为分裂属性，但是这些属性不一定是最佳分裂属性。ID3 算法只能处理离散属性，不能增量地接受训练集，每增加一次实例就抛弃原有的决策树而重新构造新的决策树，开销很大。

13.6 关联规则挖掘

13.6.1 关于关联规则挖掘

关联规则挖掘是数据挖掘中最为活跃的研究课题之一，它是从事物同时出现的现象背后发现事物之间可能存在的关联或者联系。频繁同时出现的事物之间存在某种关联关

系，因此首先要挖掘频繁地出现在数据集中的模式，即频繁项集；然后从这些频繁项集中总结事物的关联规则来指导实践。

频繁同时出现在交易数据集中的商品的集合是频繁项集。包含 k 个项的项集称为 k 项集。项集的出现频度是包含项集的事务数，简称项集的频度。如果某项集的支持度大于定义的最小支持度阈值，则该项集是频繁项集。如果顾客先购买计算机，然后购买杀毒软件，如果这种购买习惯频繁地出现(即大于定义的最小支持度阈值)在购物历史数据中，则称(计算机，杀毒软件)为一个频繁二项集。反映商品频繁关联或同时购买的购买模式可以用关联规则(association rule)的形式表示。例如，购买计算机也趋向于同时购买杀毒软件，可以用以下关联规则表示：computer->antivirus_software[support=60%；confidence=80%]。规则的支持度(support)和置信度(confidence)是兴趣度的两种度量。它们分别反映所发现规则的有用性和确定性。支持度为60%，意味着所分析的所有事务的60%显示计算机和杀毒软件被同时购买。置信度为80%，意味着购买计算机的顾客80%也购买了杀毒软件。如果一个关联规则的支持度大于系统定义的最小支持度，置信度大于系统定义的最小置信度，则称此规则为强规则。

关联规则经典算法有 Apriori 算法和 FPTree 算法，本节以 Apriori 算法为例描述关联规则挖掘的过程。

13.6.2 Apriori 算法

Apriori 算法是一种非常有影响的挖掘布尔关联规则的算法。为了尽早地消除一些完全不可能是频繁项集的集合，有两条 Apriori 性质需要遵守：任意频繁项集的非空真子集必须是频繁的；任意非频繁项集的超集一定是非频繁的。Apriori 算法关联规则挖掘分为两个阶段：第一个阶段生成频繁项集，即找到所有满足最小支持度阈值的项集；第二个阶段生成关联规则，即从频繁项集中找出所有满足最小置信度阈值的规则。

Apriori 算法使用自底向上的实现方式。首先扫描数据库找出频繁1项集的集合(L1)，基于 L1 连接来找频繁2项集的集合 L2，基于 L2 连接来找频繁3项集的集合 L3，……，采用逐层搜索的迭代方法，直到不能找到满足条件的频繁项集。然后构造各个频繁项集的关联规则，选择满足最小置信度的强关联规则即为要挖掘的关联规则。

【例 13.11】 已知原始数据集 data_set=[['l1','l2','l5'],['l2','l4'],['l2','l3'],['l1','l2','l4'],['l1','l3'],['l2','l3'],['l1','l3'],['l1','l2','l3','l5'],['l1','l2','l3']]，求其频繁项集及关联规则。

程序如下：

```
#加载数据集并返回一个数据集：一个事务列表。每个事务包含若干项
def load_data_set():
    data_set = [['l1','l2','l5'],['l2','l4'],['l2','l3'],
```

```
            ['l1','l2','l4'],['l1','l3'],['l2','l3'],
            ['l1','l3'],['l1','l2','l3','l5'],['l1','l2','l3']]
return data_set
#通过扫描数据集创建候选频繁 1项集 C1
#参数 data_set:一个事务列表。每个事务包含若干项
#返回一个包含所有频繁的候选项集的集合
def create_C1(data_set):
    C1 = set()
    for t in data_set:
        for item in t:
            item_set = frozenset([item])
            C1.add(item_set)
return C1
#判断一个频繁的候选 k项集是否满足 Apriori 属性
#参数 Ck_item:Ck 中频繁的候选 k项集 k-itemsets 候选项
#Lksub1:Lk-1,一个包含所有频繁的候选 (k-1)项集的集合
def is_apriori(Ck_item,Lksub1):
    for item in Ck_item:
        sub_Ck = Ck_item-frozenset([item])
        if sub_Ck not in Lksub1:
            return False
return True
#创建 Ck,一个包含所有频繁的候选 k项集的集合,通过 Lk-1自身的连接操作。返回
#一个包含所有频繁的候选 k项集的集合
def create_Ck(Lksub1,k):
    Ck = set()
    len_Lksub1 = len(Lksub1)
    list_Lksub1 = list(Lksub1)
    for i in range(len_Lksub1):
        for j in range(1,len_Lksub1):
            l1 = list(list_Lksub1[i])
            l2 = list(list_Lksub1[j])
            l1.sort()
            l2.sort()
            if l1[0:k-2] == l2[0:k-2]:
                Ck_item = list_Lksub1[i] | list_Lksub1[j]
                                                   #pruning
                if is_apriori(Ck_item,Lksub1):
                    Ck.add(Ck_item)
return Ck
#从 Ck执行删除策略生成 Lk
#参数
```

```
#data_set:一个事务列表。每个事务包含若干项
#Ck:一个包含所有频繁的候选 k项集的集合
#min_support:最小支持度计数
#support_data:字典。键是频繁项集,值是支持度计数
#返回 Lk: 一个包含所有频繁 k项集的集合
def generate_Lk_by_Ck(data_set,Ck,min_support,support_data):
    Lk=set()
    item_count={}
    for t in data_set:
        for item in Ck:
            if item.issubset(t):
                if item not in item_count:
                    item_count[item] =1
                else:
                    item_count[item]+=1
    t_num=float(len(data_set))
    for item in item_count:
        if (item_count[item] / t_num) >=min_support:
            Lk.add(item)
            support_data[item] =item_count[item] / t_num
    return Lk
#生成所有频繁项集。k为所有频繁项集的最大项数
def generate_L(data_set,k,min_support):
    support_data={}
    C1=create_C1(data_set)
    L1=generate_Lk_by_Ck(data_set,C1,min_support,support_data)
    Lksub1=L1.copy()
    L=[]
    L.append(Lksub1)
    for i in range(2,k+1):
        Ci =create_Ck(Lksub1,i)
        Li =generate_Lk_by_Ck(data_set,Ci,min_support,support_data)
        Lksub1 =Li.copy()
        L.append(Lksub1)
return L,support_data
#从频繁项集生成关联规则。返回 big_rule_list:包含所有大规则的列表。每个关联规则都表
#示为一个三元组
def generate_big_rules(L,support_data,min_conf):
    big_rule_list =[]
    sub_set_list =[]
    for i in range(0,len(L)):
```

```
            for freq_set in L[i]:
                for sub_set in sub_set_list:
                    if sub_set.issubset(freq_set):
                        conf=support_data[freq_set] / support_data[freq_set-sub_set]
                        big_rule=(freq_set-sub_set,sub_set,conf)
                        if conf>=min_conf and big_rule not in big_rule_list:
                            #print freq_set-sub_set," =>",sub_set,"conf: ",conf
                            big_rule_list.append(big_rule)
                sub_set_list.append(freq_set)
    return big_rule_list
#主程序
if __name__=="__main__":
    data_set=load_data_set()
    L,support_data=generate_L(data_set,k=3,min_support=0.2)
    big_rules_list=generate_big_rules(L,support_data,min_conf=0.7)
    for Lk in L:
        print("="*50)
        print("frequent"+str(len(list(Lk)[0]))+"-itemsets\t\tsupport")
        print("="*50)
        for freq_set in Lk:
            print(freq_set,support_data[freq_set])
    print()
    print("Big Rules")
    for item in big_rules_list:
        print(item[0],"=>",item[1],"conf: ",item[2]) item[1],"conf: ",item[2])
```

程序运行结果如图 13.19 所示。

习　　题

1. 描述 ID3 算法的核心思想及其优缺点。
2. 数据挖掘的聚类方法有哪些？请详细阐述之。
3. 编写程序，实现将销售单 1 转化为销售单 2。

销售单 1　销售明细如下。

```
S01 G0001
S01 G0002
S01 G0003
S02 G0004
S03 G0003
S04 G0004
S04 G0005
```

```
frequent 1-itemsets                 support
==================================================
frozenset({'l5'}) 0.2222222222222222
frozenset({'l4'}) 0.2222222222222222
frozenset({'l3'}) 0.6666666666666666
frozenset({'l1'}) 0.6666666666666666
frozenset({'l2'}) 0.7777777777777778
==================================================
frequent 2-itemsets                 support
==================================================
frozenset({'l3', 'l1'}) 0.4444444444444444
frozenset({'l5', 'l1'}) 0.2222222222222222
frozenset({'l2', 'l3'}) 0.4444444444444444
frozenset({'l2', 'l1'}) 0.4444444444444444
frozenset({'l2', 'l4'}) 0.2222222222222222
frozenset({'l5', 'l2'}) 0.2222222222222222
==================================================
frequent 3-itemsets                 support
==================================================
frozenset({'l5', 'l1', 'l2'}) 0.2222222222222222
frozenset({'l2', 'l3', 'l1'}) 0.2222222222222222

Big Rules
frozenset({'l5'}) => frozenset({'l1'}) conf:  1.0
frozenset({'l4'}) => frozenset({'l2'}) conf:  1.0
frozenset({'l5'}) => frozenset({'l2'}) conf:  1.0
frozenset({'l5', 'l2'}) => frozenset({'l1'}) conf:  1.0
frozenset({'l5', 'l1'}) => frozenset({'l2'}) conf:  1.0
frozenset({'l5'}) => frozenset({'l2', 'l1'}) conf:  1.0
```

图 13.19 Apriori 算法运行结果

销售单 2　销售明细如下。

```
S01 G0001,G0002,G0003
S02 G0004
S03 G0003
S04 G0004,G0005
```

4. 利用 K-means 算法将表 13.2 中的数据划分为两个簇。进行三次迭代，分别写出每次迭代过程中的聚类中心点和聚类结果。第一次聚类中心选取 P_1、P_2。

表 13.2 聚类数据表

	X	Y
P_1	0	0
P_2	1	2
P_3	3	1
P_4	8	8
P_5	9	10
P_6	10	7

5. 现有 A、B、C、D、E 五种商品的交易记录如表 13.3 所示，找出所有频繁项集，假设最小支持度≥50%，最小置信度≥50%。计算其各频繁项集及其关联规则。

表 13.3　商品交易记录表

交易编号	商 品 代 码
T1	A、C、D
T2	B、C、E
T3	A、B、C、E
T4	B、E

附录 A

常用字符与 ASCII 码对照表

ASCII 码由三部分组成。

第一部分为 00H～1FH，共 32 个，一般用来通信或进行控制。有些字符可显示在屏幕上，有些则无法显示在屏幕上，但可以从表 A.1 看到效果（例如换行字符、归位符）。

表 A.1　ASCII 码（一）

ASCII 码	字符	控制字符	ASCII 码	字符	控制字符
000	null	NUL	016	▶	DLE
001	☺	SOH	017	◀	DC1
002	●	STX	018	↕	DC2
003	♥	ETX	019	!!	DC3
004	◆	EOT	020	¬	DC4
005	♣	ENQ	021	§	NAK
006	♠	ACK	022	▬	SYN
007	Beep	BEL	023	↨	ETB
008	Bs	BS	024	↑	CAN
009	Tab	HT	025	↓	EM
010	换行	LP	026	→	SUB
011	（home）	VT	027	←	ESC
012	（form feed）	FF	028	∟	PS
013	回车	CR	029	↔	GS
014	♫	SO	030	▲	RS
015	☼	SI	031	▼	US

第二部分为 20H～7FH，共 96 个，除 32H 表示的空格外，其余 95 个字符用来表示阿拉伯数字、英文字母大小写和括号等符号，都可以显示在屏幕上，见表 A.2。



表 A.2　ASCII 码(二)

ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
032	(Space)	064	@	096	’
033	!	065	A	097	a
034	"	066	B	098	b
035	#	067	C	099	c
036	$	068	D	100	d
037	%	069	E	101	e
038	&	070	F	102	f
039	'	071	G	103	g
040	(	072	H	104	h
041	)	073	I	105	i
042	*	074	J	106	j
043	+	075	K	107	k
044	,	076	L	108	l
045	—	077	M	109	m
046	·	078	N	110	n
047	/	079	O	111	o
048	0	080	P	112	p
049	1	081	Q	113	q
050	2	082	R	114	r
051	3	083	S	115	s
052	4	084	T	116	t
053	5	085	U	117	u
054	6	086	V	118	v
055	7	087	W	119	w
056	8	088	X	120	x
057	9	089	Y	121	y
058	：	090	Z	122	z
059	；	091	[	123	〈
060	<	092	\	124	¦
061	=	093	]	125	〉
062	>	094	∧	126	～
063	?	095	_	127	DEL

第三部分为 80H～0FFH,共 128 个字符,一般称为扩充字符。这 128 个扩充字符是由 IBM 制定的,并非标准的 ASCII 码。这些字符是用来表示框线、音标和其他欧洲非英语系的字母,如表 A.3 所示。

表 A.3　ASCII 码（三）

ASCII 值	字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
128	Ç	160	á	192	└	224	α
129	ü	161	í	193	┴	225	β
130	é	162	ó	194	┬	226	Γ
131	â	163	ú	195	├	227	π
132	ā	164	ń	196	─	228	Ξ
133	à	165	ª	197	┼	229	σ
134	å	166	º	198	╞	230	μ
135	ç	167	ϱ	199	╟	231	τ
136	ê	168	¿	200	╚	232	Φ
137	ë	169	⌐	201	╔	233	θ
138	è	170	¬	202	╩	234	Ω
139	ï	171	1/2	203	╦	235	δ
140	î	172	1/4	204	╠	236	∞
141	ì	173	!	205	═	237	∮
142	Ã	174	《	206	╬	238	∈
143	Å	175	》	207	╧	239	∩
144	É	176	░	208	╨	240	≡
145	ac	177	▒	209	╤	241	±
146	ÅE	178	▓	210	╥	242	≥
147	ô	179	│	211	╙	243	≤
148	ö	180	╡	212	╘	244	⌠
149	ò	181	├	213	╒	245	⌡
150	û	182	╢	214	╓	246	÷
151	ù	183	╖	215	╫	247	≈
152	ÿ	184	╕	216	╪	248	。
153	Ö	185	╣	217	┘	249	▪
154	ü	186	│	218	└	250	.
155	₵	187	╗	219	█	251	√
156	£	188	╝	220	▄	252	Π
157	¥	189	╜	221	▌	253	Z
158	Pt	190	╛	222	▐	254	■
159	f	191	┐	223	▀	255	blank 'FF'

附录 B

Python 中运算符的优先级表

Python 运算符优先级如表 B.1 所示。

表 B.1　Python 运算符优先级

优先级	运算符	含　义
高	**	指数
↑	~、+、-	按位取反、正号、负号
	*、/、%、//	乘、除、取模、整除
	+、-	加法、减法
	>>、<<	右移、左移运算符
	&	按位与
	^、|	按位异或、按位或
	>、>=、<、<=	大于、大于等于、小于、小于等于
	<>、==、!=	不等于、等于、不等于
	=、+=、-=、*=、/=、%=、//=、**=	赋值、加等于、减等于、乘等于、除等于、取模等于、整除等于、幂等于
	is、is not	身份运算符
	in、not in	成员运算符
低	not、or、and	逻辑非、逻辑或、逻辑与

附录 C Python 内置函数

1. 类型转换函数

类型转换函数如表 C.1 所示。

表 C.1 类型转换函数

函　数	功 能 描 述
chr(x)	将整数 x 转换为字符
complex(real [,imag])	创建一个复数
dict(d)	创建一个字典 d
eval(str)	将字符串 str 当成有效的表达式来求值并返回计算结果
float(x)	将 x 转换为浮点数
frozenset(s)	将 s 转换为不可变集合
hex(x)	将整数 x 转换为十六进制字符串
int(x [,base])	将 x 转换为整数
list(s)	将序列 s 转换为列表
long(x [,base])	将 x 转换为长整数
oct(x)	将整数 x 转换为八进制字符串
ord(x)	返回字符 x 的 ASCII 值
repr(x)	将对象 x 转换为表达式字符串
set(s)	将 s 转换为可变集合
str(x)	将对象 x 转换为字符串
tuple(s)	将序列 s 转换为元组
unichr(x)	将一个整数 x 转换为 Unicode 字符

2. 数学函数

数学函数如表 C.2 所示。



表 C.2　数学函数

函　　数	功能描述
abs(x)	求整数 x 的绝对值
ceil(x)	返回不小于 x 的最小整数
cmp(x,y)	比较 x 和 y。如果 x>y,返回 1;如果 x==y,返回 0;如果 x<y,返回 −1
exp(x)	返回指数函数 e^x 的值
fabs(x)	求浮点数 x 的绝对值
floor(x)	返回不大于 x 的最大整数
log(x)	返回 x 的自然对数的值,即 lnx 的值
$\log_{10}$(x)	返回以 10 为底的 x 的对数
max(x1,x2,...)	返回给定参数的最大值
min(x1,x2,...)	返回给定参数的最小值
modf(x)	返回 x 的整数部分与小数部分,数值符号与 x 相同,整数部分以浮点型表示
pow(x,y)	计算 x^y 的值
round(x [,n])	返回浮点数 x 的四舍五入值,如给出 n 值,则 n 代表舍入到小数点后的位数
sqrt(x)	返回数字 x 的平方根

3. 三角函数

三角函数如表 C.3 所示。

表 C.3　三角函数

函　数	功能描述	函　数	功能描述
acos(x)	返回 x 的反余弦弧度值	hypot(x,y)	返回欧几里得范数 sqrt(x*x+y*y)
asin(x)	返回 x 的反正弦弧度值	sin(x)	返回的 x 弧度的正弦值
atan(x)	返回 x 的反正切弧度值	tan(x)	返回 x 弧度的正切值
atan2(y,x)	返回给定的 x 及 y 坐标值的反正切值	degrees(x)	将弧度转换为角度
cos(x)	返回 x 的弧度的余弦值	radians(x)	将角度转换为弧度

4. 随机函数

随机函数如表 C.4 所示。

表 C.4　随机函数

函　　数	功能描述
choice(seq)	从序列 seq 中返回随机的元素
randrange ([start,] stop [,step])	从指定范围内,按指定基数递增的集合中获取一个随机数,基数默认值为 1

续表

函　　数	功 能 描 述
random()	在[0,1)内随机生成一个实数
seed([x])	改变随机数生成器的种子 seed
shuffle(lst)	将序列的所有元素随机排序
uniform(x,y)	在[x,y]内随机生成一个实数

5. 字符串内建函数

字符串内建函数如表 C.5 所示。

表 C.5　字符串内建函数

函　　数	功 能 描 述
str.capitalize()	将字符串的第一个字母变成大写,其他字母变小写
str.center(width)	返回一个原字符串,居中显示,并使用空格填充至长度 width 的新字符串
str.count(s,begin=0,end=len(string))	返回 s 在 str 里面出现的次数。如果 begin 或 end 指定,则返回指定范围内 s 出现的次数
str.decode(encoding='UTF-8',errors='strict')	以 encoding 指定的编码格式解码字符串,errors 参数指定不同的错误处理方案
str.encode(encoding='UTF-8',errors='strict')	以 encoding 指定的编码格式编码字符串
str.expandtabs(tabsize=8)	把字符串 str 中的 tab 符号转为空格,默认的空格数 tabsize 是 8
str.find(s,begin=0,end=len(str))	检测 s 是否包含在 str 中,如果包含,则返回 s 在 str 中开始的索引值;否则返回-1
str.index(s,begin=0,end=len(str))	检测 s 是否包含在 str 中。如果包含,则返回 s 在 str 中开始的索引值;否则抛出异常
str.isalnum()	检测字符串是否由字母和数字组成。如果 str 至少有一个字符并且所有字符都是字母或数字,则返回 True;否则返回 False
str.isalpha()	检测字符串是否只由字母组成。如果 str 至少有一个字符并且所有字符都是字母,则返回 True;否则返回 False
str.isdecimal()	检查字符串是否只包含十进制字符。如果是,则返回 True;否则返回 False
str.isdigit()	检测字符串是否只由数字组成。如果 str 只包含数字则返回 True;否则返回 False
str.islower()	检测字符串是否由小写字母组成。如果 str 中包含至少一个区分大小写的字符,并且所有这些(区分大小写的)字符都是小写,则返回 True;否则返回 False
str.isnumeric()	检测字符串是否只由数字组成。如果 str 中只包含数字字符,则返回 True;否则返回 False

续表

函　　数	功 能 描 述
str. isspace()	检测字符串是否只由空格组成。如果 str 中只包含空格，则返回 True；否则返回 False
str. istitle()	检测字符串中所有的单词拼写首字母是否为大写，且其他字母为小写。如果 str 中所有的单词拼写首字母是为大写，且其他字母为小写，则返回 True；否则返回 False
str. isupper()	检测字符串中所有的字母是否都为大写。如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True；否则返回 False
str. join(seq)	将序列 seq 中的元素以指定的字符连接生成一个新的字符串
str. ljust(width)	返回一个原字符串且左对齐，并使用空格填充至长度 Width 的新字符串。如果指定的长度小于原字符串的长度，则返回原字符串
str. lstrip()	截掉字符串 str 左边的空格或指定字符
str. maketrans(intab,outtab])	创建字符映射的转换表，返回字符串转换后生成的新字符串
str. partition(s)	根据指定的分隔符 s 将字符串进行分割，返回一个三元的元组：第一个为分隔符左边的子串；第二个为分隔符本身；第三个为分隔符右边的子串
str. replace(old,new[,max])	把字符串中的 old(旧字符串)替换成 new(新字符串)，如果指定第三个参数 max，则替换不超过 max 次
str. rfind(s, begin=0, end=len(str)	返回字符串最后一次出现的位置(从右向左查询)，如果没有匹配项则返回－1
str. rindex(s, begin=0, end=len(str))	返回字符串最后一次出现的位置(从右向左查询)，如果没有匹配项则抛出异常
str. rjust(width)	返回一个原字符串且右对齐，并使用空格填充至长度 width 的新字符串
str. rpartition(s)	从后往前查找，返回包含字符串中分隔符之前、分隔符、分隔符之后的子字符串；如果没找到分隔符，返回字符串和两个空字符串
str. rstrip()	删除 str 字符串末尾的空格
str. split(s="",num=str. count(s))	以 s 为分隔符切片 str，如果 num 有指定值，则仅分隔 num 个子字符串
str. splitlines(num = str. count('\n'))	按照行分隔，返回一个包含各行作为元素的列表。如果 num 指定则仅切片 num 个行
str. startswith(s,begin=0,end=len(str))	检查字符串是否是以指定字符串 s 开头。如果是，则返回 True；否则返回 False。如果 begin 和 end 指定值，则在指定范围内检查
str. strip([s])	移除字符串头尾指定的字符 s，s 默认值是空格
str. swapcase()	对字符串的大小写字母进行转换
str. translate(s,del="")	根据 s 给出的表(包含 256 个字符)转换 str 的字符，要过滤掉的字符放到 del 参数中
str. upper()	将字符串中的小写字母转为大写字母
str. zfill(width)	返回长度为 width 的字符串，原字符串 str 右对齐，前面填充 0

6. 列表函数及方法

列表函数及方法如表 C.6 所示。

表 C.6　列表函数及方法

函　　数	功能描述
cmp(list1,list2)	比较两个列表的元素
len(list)	列表元素个数
list(seq)	将元组转换为列表
max(list)	返回列表元素最大值
min(list)	返回列表元素最小值
list.append(obj)	在列表末尾添加新的对象
list.count(obj)	统计某个元素在列表中出现的次数
list.extend(seq)	在列表末尾一次性追加另一个序列中的多个值
list.index(obj)	从列表中找出某个值第一个匹配项的索引位置
list.insert(index,obj)	将对象插入列表
list.pop(obj=list[-1])	移除列表中的一个元素(默认最后一个元素),并且返回该元素的值
list.remove(obj)	移除列表中某个值的第一个匹配项
list.reverse()	反向列表中元素
list.sort([func])	对原列表进行排序

7. 元组内置函数

元组内置函数如表 C.7 所示。

表 C.7　元组内置函数

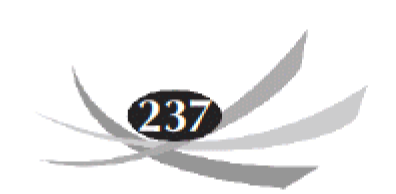
函　　数	功能描述	函　　数	功能描述
cmp(tuple1,tuple2)	比较两个元组元素	min(tuple)	返回元组中元素最小值
len(tuple)	计算元组元素个数	tuple(seq)	将列表转换为元组
max(tuple)	返回元组中元素最大值		

8. 字典内置函数及方法

字典内置函数及方法如表 C.8 所示。

表 C.8　字典内置函数及方法

函　　数	功能描述
cmp(dict1,dict2)	比较两个字典元素
len(dict)	计算字典元素个数,即键的总数



续表

函　　数	功 能 描 述
str(dict)	输出字典可打印的字符串表示
type(variable)	返回输入的变量类型,如果变量是字典就返回字典类型
dict. clear()	删除字典内所有元素
dict. copy()	返回一个字典的浅复制
dict. fromkeys(seq[,value]))	创建一个新字典,以序列 seq 中元素做字典的键,value 为字典所有键对应的初始值
dict. get(key,default=None)	返回指定键的值,如果值不在字典中返回 default 值
dict. has_key(key)	判断键是否存在于字典中。如果键在字典中,则返回 True;否则返回 False
dict. items()	以列表返回可遍历的(键,值) 元组数组
dict. keys()	以列表返回字典中所有的键
dict. setdefault (key, default = None)	判断键是否存在于字典中。如果键在字典中,则返回 True,且会添加键并将值设为 default
dict. update(dict1)	把字典 dict1 的键值对更新到 dict 中
dict. values()	以列表返回字典中的所有值

9. 时间内置函数

时间内置函数如表 C. 9 所示。

表 C. 9　时间内置函数

函　　数	功 能 描 述
time. altzone()	返回格林尼治西部的夏令时地区的偏移秒数。如果该地区在格林尼治东部会返回负值(如西欧,包括英国)。对夏令时启用地区才能使用
time. asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"(2008 年 12 月 11 日 周二 18 时 07 分 14 秒)的 24 个字符的字符串
time. clock()	用以浮点数计算的秒数返回当前的 CPU 时间。用来衡量不同程序的耗时,比 time. time()更有用
time. ctime([sec])	把一个时间戳 sec(按秒计算的浮点数)转换为 time. asctime()的形式
time. gmtime([sec])	将一个时间戳转换为 UTC 时区(0 时区)的 struct_time,可选的参数 sec 表示自 1970-1-1 以来的秒数
time. localtime([sec])	格式化时间戳为本地的时间

续表

函　　数	功 能 描 述
time. mktime(tupletime)	接收 struct_time 对象作为参数,返回用秒数来表示时间的浮点数
time. sleep(sec)	推迟调用线程的运行
time. strftime(fmt[,tupletime])	将 gmtime()或 localtime()返回的时间元组转换为 fmt 参数指定的字符串
time. strptime(str,fmt='%a%b %d %H:%M:%S %Y')	根据 fmt 的格式把一个时间字符串解析为时间元组
time. time()	返回当前时间的时间戳(1970 年后到当前时间经过的浮点秒数)
time. tzset()	根据环境变量 timezone 重新初始化时间相关设置

10. os 模块关于目录/文件操作的常用函数

os 模块关于目录/文件操作的常用函数如表 C. 10 所示。

表 C. 10　os 模块关于目录/文件操作的常用函数

函　数　名	功 能 描 述
getcwd()	显示当前工作目录
chdir(newdir)	改变当前工作目录
listdir(path)	列出指定目录下所有的文件和目录
mkdir(path)	创建单级目录
makedirs(path)	递归地创建多级目录
rmdir(path)	删除单级目录
removedirs(path)	递归地删除多级空目录,从子目录到父目录逐层删除,遇到目录非空则抛出异常
rename(old,new)	将文件或目录 old 重命名为 new
remove(path)	删除文件
stat(file)	获取文件 file 的所有属性
chmod(file)	修改文件权限
system(command)	执行操作系统命令
exec()或 execvp()	启动新进程
osspawnv()	在后台执行程序
exit()	终止当前进程

11. os. path 模块中常用的函数

os. path 模块中常用的函数如表 C. 11 所示。

表 C.11　os.path 模块中常用的函数

函　数　名	功能描述
split(path)	分离文件名与路径
splitext(path)	分离文件名与扩展名
abspath(path)	获得文件的绝对路径
dirname(path)	去掉文件名，只返回目录路径
getsize(file)	获得指定文件的大小
getatime(file)	返回指定文件最近的访问时间
getctime(file)	返回指定文件的创建时间
getmtime(file)	返回指定文件最新的修改时间
basename(path)	去掉目录路径，只返回路径中的文件名
exists(path)	判断文件或者目录是否存在
islink(path)	判断指定路径是否绝对路径
isfile(path)	判断指定路径是否存在且是一个文件
isdir(path)	判断指定路径是否存在且是一个目录
isabs(path)	判断指定路径是否存在且是一个
walk(path)	搜索目录下的所有文件

参 考 文 献

[1] William F Punch,Richard Enbody. Python 入门经典[M]. 北京：机械工业出版社,2012.
[2] Magnus Lie Hetland. Python 基础教程[M]. 2 版. 北京：人民邮电出版社,2010.
[3] 赵家刚,狄光智,吕丹桔. 计算机编程导论——Python 程序设计[M]. 北京：人民邮电出版社,2013.
[4] 董付国. Python 程序设计[M]. 2 版. 北京：清华大学出版社,2015.
[5] 刘卫国. Python 语言程序设计[M]. 北京：电子工业出版社,2016.